



Detailed Explanation of Visual C++ Digital Image Processing with Typical Cases

# Visual C++

## 数字图像

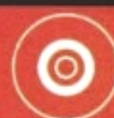
## 案例详解

沈晶 刘海波 周长建 等编著

Detailed Explanation of Visual C++ Digital Image Processing with Typical Cases



机械工业出版社  
China Machine Press



CD-ROM

# Visual C++

## 数字图像处理典型案例详解

沈晶 刘海波 周长建◎等编著

Detailed Explanation of Visual C++ Digital Image Processing with Typical Cases



机械工业出版社  
China Machine Press



本书以 Visual C++ 数字图像处理技术为主线, 结合典型的图像系统开发案例, 按照从理论、设计到实现的过程进行剖析讲解。案例从应用角度涉及娱乐、文化、医疗、交通、遥感、安防、司法等多个典型应用领域, 从技术角度涉及数字图像的文件读写、显示、编辑、滤镜增效、压缩编解码、几何变换、灰度变换、色彩空间变换、特征变换、增强、分割、复原、配准、检索、重建、形态学处理、运动目标检测、跟踪、识别等, 几乎涵盖了数字图像处理的整个技术领域及部分模式识别内容, 同时还介绍了 OpenCV 和 VTK 等开发环境及其与 Visual C++ 联合开发的实用技术。在每个案例的最后, 还与读者分享了开发经验。本书配有书中全部案例的完整源程序, 便于读者学习和在实际开发中使用。

本书适合从事计算机视觉、数字图像处理、模式识别相关工作的研究人员、工程技术人员, 以及相关专业的教师和学生阅读参考。

封底无防伪标均为盗版

版权所有, 侵权必究

本书法律顾问 北京市展达律师事务所

## 图书在版编目 (CIP) 数据

Visual C++ 数字图像处理典型案例详解 / 沈晶等编著. —北京: 机械工业出版社, 2012.6

ISBN 978-7-111-38871-5

I. V… II. 沈… III. C 语言—数字图像处理—程序设计 IV. ①TP391.41 ②TP312

中国版本图书馆 CIP 数据核字 (2012) 第 131066 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 陈佳媛

藁城市京瑞印刷有限公司印刷

2012 年 7 月第 1 版第 1 次印刷

186mm×240mm • 30.75 印张

标准书号: ISBN 978-7-111-38871-5

ISBN 978-7-89433-501-2 (光盘)

定价: 69.00 元 (附光盘)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com

# 前 言

---

“外面的世界很精彩，外面的世界很无奈”，如果用计算机的“眼睛”去看世界，数字图像处理技术既能帮你留下这份精彩，又能帮你应对那份无奈。

数字图像处理 (Digital Image Processing) 是通过计算机对图像进行去除噪声、增强、复原、分割、提取特征等处理的方法和技术。图像是人类获取和交换信息的主要来源，正因如此，数字图像处理技术便自然而然地走进了人类生产生活的方方面面。

我们在 2010 年出版了《Visual C++ 数字图像处理技术详解》<sup>①</sup>一书，介绍了数字图像处理基本算法的原理及其在 Visual C++ 平台上的编程实现方法，是围绕算法组织内容的。

本书则不同，是围绕项目案例组织内容的，从娱乐、文化、医疗、交通、遥感、安防、司法等领域精选了一些典型案例，对每个案例的技术原理、系统结构、主要流程及其 Visual C++ 程序源码进行了详细的解读。本书能够帮助读者学习和理解那些数字图像处理基本算法是如何构成一个能处理实际问题的应用系统的。

## 本书特点

- 案例丰富，工程性强

本书的重点不是讲解数字图像处理基本算法的编程实现问题，而是围绕工程项目案例来解读如何用数字图像处理基本算法构筑应用程序的问题。书中精选了 13 个典型的数字图像处理系统案例，这些案例涉及娱乐、文化、医疗、交通、遥感、安防、司法等多个应用领域，各个领域的读者从书中都可以找到自己熟悉的案例。读者可以在学习这些案例的基础上，快速掌握数字图像处理工程项目的开发方法。

- 解读深入，技术性强

对每一个案例，都深入剖析了其中的核心技术原理，对其技术路线、计算方法、处理过程都

---

① 书号为 978-7-111-30420-3，由机械工业出版社出版。

进行了深入的解读。技术内容涉及数字图像的文件读写、显示、编辑、滤镜增效、压缩编解码、几何变换、灰度变换、色彩空间变换、特征变换、增强、分割、复原、配准、检索、重建、形态学处理、运动目标检测、跟踪、识别等，同时还介绍了 OpenCV 和 VTK 等开发环境及其与 Visual C++ 联合开发的实用技术。对部分功能的实现，书中介绍了多种技术途径，以帮助读者开阔视野，进而在工程实践中能有更多的选择。

- 代码完整，实用性强

书中对每个案例都介绍了其主要功能、总体结构和主要流程，并对核心代码的实现思路进行了详细的说明，关键代码均给出了详尽的注释。配套光盘中给出了完整的源代码，读者可以直接编译运行，边做边学。在实际开发中，也可以直接重用这些代码，快速开发出产品原型，不必从零开始。为了便于读者学习和使用这些代码，全部代码都统一为 Visual C++ 2008 版本，以免读者进行版本迁移的麻烦。

- 抛砖引玉，启发性强

通过每个案例的学习，读者都可以搞清楚其中最基本的技术原理和编程实现思路。在每个案例最后都分享了开发者的经验，总结了数字图像处理项目开发过程中经常遇到的问题、可能出现的问题、应该特别注意的问题、需要进一步考虑的问题等，并探讨了这些问题的解决方案或应对思路。读者可以在书中案例的基础之上充分发挥，结合实际需求和项目背景，开发出更多、更实用、更漂亮的数字图像处理应用系统。

## 主要内容

本书共 14 章。各章的主要内容如下。

第 1 章主要介绍数字图像处理系统开发的基本知识，侧重介绍常用的开发平台，包括 Visual C++、OpenCV 和 VTK。通过本章读者可以了解这些开发平台的特点、版本演进过程及其用于开发数字图像处理程序的方法。

第 2 章详细解读一个绘图板程序，包括图形设备接口技术，几何图形绘制技术，绘图板的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解类似 Windows 画图程序的技术原理及开发方法。

第 3 章详细解读一个图片浏览器程序，包括图像文件的编解码技术，图像的几何变换技术，图像的切换特效技术，常见的图像格式分析，图片浏览器的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解类似 ACDSee 软件的技术原理及开发方法。

第 4 章详细解读一个图像编辑器程序，包括图像灰度变换增强技术，直方图增强技术，平滑去噪技术，图像锐化技术，模糊复原技术，彩色增强技术，滤镜技术，图像编辑器的功能描述、



总体结构、主要流程和编程实现。通过本章读者可以深入了解类似 Photoshop 软件的技术原理及开发方法。

第 5 章详细解读一个 CT 图像重建系统,包括三维可视化技术,图像重建技术,CT 图像重建系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解三维图像重建技术原理及其在医学领域的应用。

第 6 章详细解读一个数字图像水印系统,包括数字图像水印嵌入与提取技术,数字图像水印系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解数字图像水印技术原理及其在版权保护领域的应用。

第 7 章详细解读一个遥感图像配准系统,包括遥感图像几何校正技术、辐射校正技术、增强技术、配准技术、遥感图像配准系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解图像配准技术原理及其在遥感领域的应用。

第 8 章详细解读一个图像检索系统,包括图像特征提取技术,相似度计算技术,图像检索系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解图像检索系统的技术原理及开发方法。

第 9 章详细解读一个细胞检测与计数系统,包括显微图像去噪技术,颜色空间及其转换技术,阈值分割技术, Blob 分析技术,边缘提取技术,细胞检测与计数系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解图像分割相关的技术原理及在医疗领域的应用。

第 10 章详细解读一个指纹提取与识别系统,包括指纹图像场及其分割技术,指纹图像增强与细化技术,指纹局部特征点提取技术,指纹匹配技术,指纹提取与识别系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解指纹图像处理与分析的技术原理及其在司法领域的应用。

第 11 章详细解读一个人脸检测与识别系统,包括人脸检测及定位技术,人脸特征提取技术,人脸识别技术,人脸检测与识别系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解人脸图像处理与分析的技术原理及其在安防领域的应用。

第 12 章详细解读一个运动车辆检测跟踪系统,包括运动目标检测技术,运动目标跟踪技术,运动车辆检测跟踪系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解运动目标检测跟踪的技术原理及其在智能交通领域的应用。

第 13 章详细解读一个车型识别系统,包括基于背景去除的目标分割技术,车型特征提取技术,车型分类识别技术,车型识别系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可以深入了解车型识别的技术原理及其在车辆收费管理领域的应用。

第 14 章详细解读一个车牌识别系统,包括车牌图像预处理技术,车牌定位技术,车牌字符分割与识别技术,车牌识别系统的功能描述、总体结构、主要流程和编程实现。通过本章读者可

以深入了解车牌识别系统的技术原理及开发方法。

## 读者对象

- 计算机视觉系统开发人员。
- 数字图像处理系统开发人员。
- 数字图像处理相关领域科研人员。
- 数字图像处理编程爱好者。
- Visual C++程序设计爱好者。
- 高等院校相关的教师和学生。

## 本书光盘

- 程序源代码：包含书中全部典型案例的完整源代码及测试用的图像文件，读者可以对照书中的讲解对程序源代码展开深入研究。
- 开源软件包：包含书中案例要用到的 VTK、CMake 和 OpenCV 开源软件包，读者可以直接安装使用。

本书主要由沈晶、刘海波、周长建编著，参加本书编著、案例开发和资料整理的还有刘萌、崔莹、于义雪、党银强、岳振勋、刘英波、唐坤、吴艳霞、郭耸、于化龙、宋锋、林玉娥、朱长明、宋一兵、管殿柱等。本书得到中央高校基本科研业务费专项资金和哈尔滨工程大学国家大学科技园基金资助。

感谢您选择了本书，希望我们的努力对您的工作和学习有所帮助，也希望您把对本书的意见和建议告诉我们。

零点工作室网站地址：[www.zerobook.net](http://www.zerobook.net)

零点工作室联系信箱：[gdz\\_zero@126.com](mailto:gdz_zero@126.com)

零点工作室

2012 年 4 月

# 目 录

---

前言

- 第 1 章 数字图像处理软件开发概述..... 1
  - 1.1 Visual C++ ..... 1
    - 1.1.1 Visual C++概述..... 2
    - 1.1.2 Visual C++处理数字图像的基本方法..... 2
  - 1.2 OpenCV ..... 4
    - 1.2.1 OpenCV 概述 ..... 5
    - 1.2.2 在 Visual C++中使用 OpenCV..... 6
  - 1.3 VTK ..... 7
    - 1.3.1 VTK 概述..... 7
    - 1.3.2 在 Visual C++中使用 VTK ..... 7
  - 1.4 经验分享..... 15
- 第 2 章 绘图板..... 16
  - 2.1 核心技术原理..... 16
    - 2.1.1 图形设备接口技术 ..... 16
    - 2.1.2 几何图形绘制技术 ..... 17
  - 2.2 系统功能..... 22
    - 2.2.1 功能描述..... 22
    - 2.2.2 界面效果..... 22
  - 2.3 系统结构与流程..... 23
    - 2.3.1 总体结构..... 23



2.3.2	主要流程	23
2.4	编程实现	24
2.4.1	绘图板初始化	24
2.4.2	图形绘制	26
2.4.3	图形保存	41
2.5	经验分享	44
第 3 章	图片浏览器	46
3.1	核心技术原理	46
3.1.1	图像文件的编解码技术	46
3.1.2	图像的几何变换技术	48
3.1.3	图像的切换特效技术	55
3.2	常见的图像格式分析	58
3.2.1	BMP 图像	58
3.2.2	PCX 图像	61
3.2.3	TGA 图像	64
3.2.4	JPEG 图像	70
3.2.5	GIF 图像	71
3.3	系统功能	74
3.3.1	功能描述	75
3.3.2	界面效果	75
3.4	系统结构与流程	76
3.4.1	总体结构	76
3.4.2	主要流程	76
3.5	编程实现	78
3.5.1	LanImage 类	78
3.5.2	BMP 解码/编码模块	80
3.5.3	PCX 解码/编码模块	81
3.5.4	TGA 解码/编码模块	84
3.5.5	图像显示模块	87
3.5.6	全屏浏览模块	88
3.5.7	图像缩放模块	90

3.5.8 图像旋转模块 .....	91
3.5.9 图像特效显示模块 .....	93
3.5.10 图像镜像模块 .....	95
3.5.11 图像转置模块 .....	96
3.6 经验分享 .....	97
<b>第 4 章 图像编辑器</b> .....	<b>101</b>
4.1 核心技术原理 .....	101
4.1.1 灰度变换增强技术 .....	101
4.1.2 直方图增强技术 .....	103
4.1.3 平滑去噪技术 .....	105
4.1.4 图像锐化技术 .....	107
4.1.5 模糊复原技术 .....	109
4.1.6 彩色增强技术 .....	110
4.1.7 滤镜技术 .....	111
4.2 系统功能 .....	115
4.2.1 功能描述 .....	115
4.2.2 界面效果 .....	116
4.3 系统结构与流程 .....	116
4.3.1 总体结构 .....	116
4.3.2 主要流程 .....	116
4.4 编程实现 .....	118
4.4.1 灰度变换增强模块 .....	118
4.4.2 直方图增强模块 .....	121
4.4.3 平滑去噪模块 .....	125
4.4.4 图像锐化模块 .....	133
4.4.5 彩色增强模块 .....	135
4.4.6 模糊复原模块 .....	136
4.4.7 滤镜效果模块 .....	139
4.5 经验分享 .....	144
<b>第 5 章 CT 图像重建系统</b> .....	<b>145</b>
5.1 核心技术原理 .....	145

5.1.1	三维可视化技术	146
5.1.2	图像重建技术	153
5.2	系统功能	158
5.2.1	功能描述	158
5.2.2	界面效果	159
5.3	系统结构与流程	160
5.3.1	总体结构	160
5.3.2	主要流程	160
5.4	编程实现	163
5.4.1	圆锥体 CT 图像重建系统	163
5.4.2	头部切片 CT 图像重建系统	165
5.5	经验分享	172
第 6 章	数字图像水印系统	173
6.1	核心技术原理	173
6.1.1	图像水印嵌入技术	174
6.1.2	图像水印提取技术	182
6.2	系统功能	186
6.2.1	功能描述	186
6.2.2	界面效果	187
6.3	系统结构与流程	188
6.3.1	总体结构	188
6.3.2	主要流程	188
6.4	编程实现	192
6.4.1	不带嵌入因子的加性规则算法实现	192
6.4.2	最低有效位算法实现	201
6.5	经验分享	203
第 7 章	遥感图像配准系统	204
7.1	核心技术原理	204
7.1.1	遥感图像几何校正技术	205
7.1.2	遥感图像辐射校正技术	208
7.1.3	遥感图像增强技术	208



7.1.4	遥感图像配准技术 .....	209
7.2	系统功能 .....	210
7.2.1	功能描述 .....	210
7.2.2	界面效果 .....	210
7.3	系统结构与流程 .....	212
7.3.1	总体结构 .....	212
7.3.2	主要流程 .....	213
7.4	编程实现 .....	213
7.4.1	CDib 类 .....	213
7.4.2	几何校正模块 .....	216
7.4.3	遥感图像增强模块 .....	228
7.4.4	遥感图像配准模块 .....	242
7.5	经验分享 .....	246
第 8 章	图像检索系统 .....	248
8.1	核心技术原理 .....	248
8.1.1	图像特征提取技术 .....	248
8.1.2	相似度计算技术 .....	257
8.2	系统功能 .....	258
8.2.1	功能描述 .....	258
8.2.2	界面效果 .....	259
8.3	系统结构与流程 .....	261
8.3.1	总体结构 .....	261
8.3.2	主要流程 .....	261
8.4	编程实现 .....	262
8.4.1	系统设置模块 .....	262
8.4.2	图像检索模块 .....	265
8.5	经验分享 .....	281
第 9 章	细胞检测与计数系统 .....	282
9.1	核心技术原理 .....	282
9.1.1	显微图像去噪技术 .....	282
9.1.2	颜色空间及其转换技术 .....	283

9.1.3	阈值分割技术	286
9.1.4	Blob 分析技术	287
9.1.5	边缘提取技术	290
9.2	系统功能	291
9.2.1	功能描述	291
9.2.2	界面效果	292
9.3	系统结构与流程	292
9.3.1	总体结构	292
9.3.2	主要流程	293
9.4	编程实现	293
9.4.1	图像平滑模块	293
9.4.2	HSI 阈值选取模块	295
9.4.3	梯度修正模块	299
9.4.4	填充孔洞模块	301
9.4.5	腐蚀模块	305
9.4.6	边界生成模块	306
9.4.7	查找中心点和修正模块	309
9.5	经验分享	320
第 10 章	指纹提取与识别系统	321
10.1	核心技术原理	321
10.1.1	指纹图像场及其分割技术	321
10.1.2	指纹图像增强技术	325
10.1.3	指纹图像细化技术	327
10.1.4	指纹局部特征点提取技术	329
10.1.5	指纹匹配技术	331
10.2	系统功能	334
10.2.1	功能描述	334
10.2.2	界面效果	334
10.3	系统结构与流程	335
10.3.1	总体结构	335
10.3.2	主要流程	335

10.4	编程实现	335
10.4.1	指纹图像分割模块	335
10.4.2	指纹图像增强模块	337
10.4.3	指纹图像二值化模块	339
10.4.4	细化模块	341
10.4.5	特征点提取模块	344
10.4.6	指纹图像比对模块	348
10.5	经验分享	362
第 11 章	人脸检测与识别系统	364
11.1	核心技术原理	364
11.1.1	人脸检测及定位技术	364
11.1.2	人脸特征提取技术	369
11.1.3	人脸识别技术	371
11.2	系统功能	372
11.2.1	功能描述	373
11.2.2	界面效果	373
11.3	系统结构与流程	373
11.3.1	总体结构	374
11.3.2	主要流程	374
11.4	编程实现	375
11.4.1	人脸检测模块	375
11.4.2	人脸定位模块	378
11.4.3	人脸特征点提取模块	382
11.4.4	人脸匹配模块	396
11.5	经验分享	398
第 12 章	运动车辆检测跟踪系统	399
12.1	核心技术原理	399
12.1.1	运动目标检测技术	399
12.1.2	运动目标跟踪技术	402
12.2	系统功能	404
12.2.1	功能描述	404



12.2.2	界面效果	404
12.3	系统结构与流程	406
12.3.1	总体结构	406
12.3.2	主要流程	406
12.4	编程实现	407
12.4.1	变量定义模块	408
12.4.2	文件打开模块	408
12.4.3	背景提取模块	409
12.4.4	车辆跟踪与检测模块	410
12.4.5	轨迹绘制模块	412
12.5	经验分享	413
第 13 章	车型识别系统	414
13.1	核心技术原理	414
13.1.1	基于背景去除的目标分割技术	414
13.1.2	车型特征提取技术	419
13.1.3	车型分类识别技术	420
13.2	系统功能	423
13.2.1	功能描述	423
13.2.2	界面效果	423
13.3	系统结构与流程	424
13.3.1	总体结构	424
13.3.2	主要流程	424
13.4	编程实现	425
13.4.1	变量定义模块	425
13.4.2	图像显示模块	426
13.4.3	载入图像模块	427
13.4.4	车辆提取模块	427
13.4.5	轮廓提取模块	431
13.4.6	车型识别模块	431
13.5	经验分享	433

第 14 章 车牌识别系统 .....	434
14.1 核心技术原理 .....	434
14.1.1 车牌图像预处理技术 .....	434
14.1.2 车牌定位技术 .....	437
14.1.3 车牌字符分割技术 .....	438
14.1.4 车牌字符识别技术 .....	440
14.2 系统功能 .....	441
14.2.1 功能描述 .....	442
14.2.2 界面效果 .....	442
14.3 系统结构与流程 .....	442
14.3.1 总体结构 .....	442
14.3.2 主要流程 .....	442
14.4 编程实现 .....	442
14.4.1 自定义函数模块 .....	442
14.4.2 车牌提取模块 .....	446
14.4.3 倾斜校正模块 .....	450
14.4.4 字符分割模块 .....	452
14.4.5 字符归一化模块 .....	456
14.4.6 字符细化模块 .....	456
14.4.7 字符特征提取模块 .....	467
14.4.8 车牌字符识别模块 .....	468
14.5 经验分享 .....	474
参考文献 .....	475

# 第 1 章 数字图像处理软件开发概述

“心有多大，舞台就有多大。”开发数字图像处理软件，需要想象力，而要把想象变成现实，则需要得心应手的开发平台。

目前常用于开发数字图像处理软件的平台可以粗略地分为通用平台和专用平台两大类。通用平台不是专门针对数字图像处理软件开发而打造的，但是却可以用于开发数字图像处理系统，如 Visual C++、Matlab 等。为了提高开发效率，通用平台往往结合专门的软件包（如 OpenCV、VTK 等）或工具箱（如 Matlab 中的 Image Processing 工具箱等）进行数字图像处理软件开发。专用平台则是专为数字图像处理或更进一步的机器视觉系统开发量身定制的，这类开发平台中均内置了大量专门用于数字图像处理的数据结构、对象、函数或组件模块，可供开发人员方便地调用，在更高的层次上进行数字图像处理系统的开发，如 Halcon、VisionPro 等。本章主要介绍本书将要用到的几种开发平台及其配置和使用方法。

## 本章要点

- Visual C++处理数字图像的基本方法
- 在 Visual C++中使用 OpenCV
- 在 Visual C++中使用 VTK

## 1.1 Visual C++

“工欲善其事，必先利其器。”Visual C++便是众多开发工具中的“一把所向披靡的利器”。Visual C++（简称 VC）是 Microsoft 公司的 Visual Studio 开发工具箱中的一个 C++程序开发环境。自诞生以来，凭借着 C++语言的强大威力、开发环境的良好支持，以及与 Windows 操作系统的“血缘”关系，一直是 Windows 操作系统环境下最主要的开发工具之一。使用 VC 可以完成各种各样应用程序的开发，从底层软件到上层直接面向用户的软件，而且用 VC 开发出的产品与 Windows 操作系统最具“亲和力”。掌握了 VC，就等于进入了 Windows 编程的自由王国。VC 在数字图像处理软件开发中也占据着极其重要的地位。

## 第1章


### 1.1.1 Visual C++概述

VC 是一个面向对象的可视化集成开发系统,它不但具有程序框架自动生成、灵活方便的类管理、代码编写和界面设计集成交互操作、可开发多种程序等优点,而且通过简单的设置就可使其生成的程序框架支持数据库接口、OLE2、WinSock 网络、3D 控制界面。VC 以拥有语法高亮、智能感知(在编辑环境中,光标悬停在函数上时显示类定义和注释,键入函数或属性名时可以自动完成这些名称的输入)、高级除错、最小重建及累加链接功能而著称,这些特色功能有助于缩短程序编辑、编译及链接的时间花费,在大型软件开发中效果尤其显著。

VC 已历经数个版本,伴随着 Microsoft .NET 计划的展开,又诞生了一系列 VC.NET 版本,从当年的 1.0 版本到现在最新的 VC 2010 (10.0 版本),VC 在界面、功能、库支持方面都增强了很多。

VC 1.0 是 Microsoft 公司于 1993 年推出的,它集成了 MFC 2.0 (MFC 是一个以 C++ 类的形式封装了 Windows API 的基础类库,其中包含一个应用程序框架、大量 Windows 句柄封装类和很多 Windows 的内建控件和组件的封装类,1992 年随微软的 Microsoft C/C++ 7.0 编译器发布,可以减少应用程序开发人员的工作量),可算是 Microsoft C/C++ 7.0 的更新版本。VC 1.5 集成了 MFC 2.5,增加了 OLE 2.0 和支持 MFC 的 ODBC。VC 2.0 集成了 MFC 3.0。VC 4.0 集成了 MFC 4.0,这个版本是专门为 Windows 95 以及 Windows NT 设计的。由于 VC 2.0 是在 Windows 95 之前发布的,且发布时间与 Windows 95 非常接近,当 Windows 95 发布时,VC 4.0 也已经发布了,很多程序员直接从 1.x 过渡到 4.0,把 2.x 跳过去了,所以 VC 2.0 的应用并不广泛。VC 5.0 集成了 MFC 4.21,是 4.2 版以来比较大的一次升级。VC 6.0 集成了 MFC 6.0,于 1998 年发布,发布至今一直被广泛地用于大大小小的项目开发。VC.NET 2002 (VC 7.0) 于 2002 年发布,集成了 MFC 7.0,支持 .NET 1.0,支持链接时代码生成和调试执行时检查。VC.NET 2003 (VC 7.1) 集成了 MFC 7.1,于 2003 年发布,支持 .NET 1.1。VC 2005 (VC 8.0) 集成了 MFC 8.0,于 2005 年发布,支持 .NET 2.0,该版本引进了对 C++/CLI 语言和 OpenMP 的支持。VC 2008 (VC 9.0) 集成了 MFC 9.0,于 2007 年发布,支持 .NET 3.5。VC 2010 (VC 10.0) 于 2010 年发布,集成了 MFC 10.0,支持 .NET 4.0,支持 C++0x 新标准。

---

 在数字图像处理领域,积累了大量采用 VC 6.0 开发的应用软件。尽管 VC 6.0 版本代码可以自动迁移到 VC.NET 版本,但由于 VC.NET 各版本与 VC 6.0 并不完全兼容,有时迁移后并不能直接编译通过,还需要对源程序做必要的修改。

---

### 1.1.2 Visual C++处理数字图像的基本方法

数字图像文件的格式多达几十种,但多数都是经过压缩的,不便于直接处理。数字图像处理时最常使用的文件格式是未经压缩的位图图像 (BMP 图像文件),其他格式的压缩图像一般都要

先解压缩成位图图像再进行处理。本节主要介绍用 VC 处理位图的基本方法。

Windows 中的位图有 DDB (Device-dependent Bitmap) 和 DIB (Device-independent Bitmap) 两种格式, 它们的文件扩展名都是 “.bmp”。

DDB 位图又叫 GDI (Graphics Device Interface, 图形设备接口) 位图, 它是一种 GDI 对象, 用句柄 HBITMAP 来操作, 或者用 MFC 类 CBitmap 的成员函数来操作。在 CBitmap 中包含一种和 Windows 的 GDI 模块相关的 Windows 数据结构, 该数据结构是与设备相关的, DDB 位图只能显示在颜色模式与其匹配的显示设备上。

DIB 位图则能保证用某个应用程序创建的位图图像可以被其他应用程序装载且显示效果相同。DIB 位图的与设备无关性主要体现在以下两个方面: 1) DIB 的颜色模式与设备无关。例如, 一个 256 色的 DIB 既可以在真彩色显示模式下使用, 也可以在 16 色模式下使用。2) 256 色 (含) 以下的 DIB 拥有自己的颜色表, 像素的颜色独立于系统调色板。

由于 DIB 不依赖于具体设备, 因此可以用来永久性地保存图像, 通常以后缀为 “.bmp” 的文件形式保存在磁盘中或作为资源存在于程序的 EXE 或 DLL 文件中, 所以, 需要保证图像的显示效果时, 要使用 DIB 位图。DDB 位图在匹配的显示设备上的显示速度要比 DIB 快, 因此, 当性能是主要问题时, 应该采用 DDB 位图。

在 VC 中, 可以通过 CBitmap 类、Image 类、Bitmap 类和 CImage 类来处理位图, 但不同的类所能处理的位图种类不同, 不同的 VC 版本对上述类的支持也不同。

CBitmap 类继承自 CGdiObject, 是封装了 GDI 的位图, 提供成员函数来操纵位图。在使用 CBitmap 对象之前需要先构造 CBitmap 对象, 再用其初始化成员函数与一个位图句柄关联起来, 然后便可以调用该 CBitmap 对象的其他成员函数了。CBitmap 类主要用于处理 DDB 位图, 封装了与 DDB 位图操作函数相关的数据结构和操作函数。结构体 BITMAP 定义了 DDB 位图的类型、宽度、高度、颜色和像素值。CBitmap 的 LoadBitmap、CreateCompatibleBitmap、SetBitmapBits、GetBitmap 等成员函数定义了对 DDB 位图的装载、创建、设定位值和属性查询等操作。创建或装入内存的位图必须用 CDC::SelectObject 函数来将其选入设备上下文中, 然后用 CDC 的 BitBlt (或 StretchBlt) 函数显示出来, 该函数把源设备上下文中的位图复制到本身的设备上下文中, 两个设备上下文可以是内存设备上下文, 也可以是同一个设备上下文。StretchBlt 函数必要时可按目标设备设置的模式进行图像的拉伸或压缩。


在 VC.NET 之前, MFC 未提供现成的类来封装 DIB, 这给 MFC 用户带来很多不便。因为用户要想使用 DIB, 首先应该了解 DIB 的结构。DIB 的颜色信息存储在自己的颜色表中, 程序一般要根据颜色表为 DIB 创建逻辑调色板。在输出一幅 DIB 之前, 程序应该将其逻辑调色板选入到相关的设备上下文中并实现到系统调色板中, 然后再调用相关的 GDI 函数 (如::SetDIBitsToDevice 或::StretchDIBits) 输出 DIB。在输出过程中, GDI 函数会把 DIB 转换成 DDB: 先将 DIB 的颜色

格式转换成与输出设备相同的颜色格式，再将 DIB 像素的逻辑颜色索引转换成系统调色板索引。由于 MFC 未提供一个封装好的易用的 DIB 类，用户在使用 DIB 时将面临繁重的 Windows API 编程任务，所以传统的图像处理方法中一般都会把这些 Win32 SDK 中操作 DIB 位图的 API 封装起来并自定义为一个通用的类来使用，以减轻后续算法编写中的编程负担。

在 VC.NET 版本中，GDI+ 的 Image 类封装了对 BMP、GIF、JPEG、PNG、TIFF、WMF 和 EMF 图像文件的调入、显示、格式转换以及简单处理（如缩放、旋转、拉伸等）的功能。而 Bitmap 类（注意不是结构 BITMAP）是从 Image 类继承的一个图像类（另一个从 Image 继承的类是 Metafile 类），它封装了 Windows 位图操作的常用功能。例如，Bitmap::SetPixel 和 Bitmap::GetPixel 分别用来对位图进行读写像素操作，从而可以为图像的柔化和锐化处理提供一种可能。这些功能和 MFC 的新类 CImage 功能基本一样，如果仅用于图像的读取与显示，用 Bitmap 类或 Image 类是不错的选择，如果是做图像处理，则 CImage 可能更符合 MFC 程序员的编程习惯。

CImage 类是 VC.NET 中 MFC 和 ATL 共享的新类，它能从外部磁盘中调入一个 BMP、JPEG、GIF 或 PNG 格式的图像文件加以显示，而且这些文件格式可以相互转换。CImage 既能处理 DIB 位图，也能处理非 DIB 位图，可以用 Create 或 Load 方法来创建或加载 DIB 位图的 CImage 对象，也可以使用 Attach 方法把一个非 DIB 位图连接给 CImage 对象，但不能使用下列方法（这些方法仅支持 DIB 位图）：GetBits、GetColorTable、GetMaxColorTable、Entries、GetPitch、GetPixelAddress、IsIndexed、SetColorTable。要确定一个连接的位图是否是一个 DIB 位图，可用成员函数 IsDibSection 来判断。CImage 提供了 HBITMAP 操作符，因此用作参数的 HBITMAP，都可以用 CImage 来替代。

由于在不同的 Windows 操作系统中 CImage 的某些性能是不一样的，因此在使用时要特别注意。例如，CImage::PlgBlt 和 CImage::MaskBlt 只能在 Windows NT 4.0 或更高版本中使用，但不能运行在 Windows 95/98 应用程序中。CImage::AlphaBlend 和 CImage::TransparentBlt 也只能在 Windows 2000/98 或更高版本中使用。即使是在 Windows 2000 上运行程序，也必须将 stdafx.h 文件中的 WINVER 和 \_WIN32\_WINNT 的预定义修改成 0x0500 才能正常使用。CImage 可以在 MFC 或 ATL 中使用。当使用 CImage 创建一个项目时，必须包含 atimage.h 文件。

 由于编程习惯或版本自动迁移等原因，不少在 VC.NET 平台上开发的程序依然使用自定义的 DIB 类来处理 DIB 位图，而没有使用 CImage 类（本书的案例中也存在这样的现象），希望 VC 图像处理编程的初学者不要被这种现象所误导，应该更多地重视 CImage 类的使用。

## 1.2 OpenCV

牛顿说：“如果说我能看得更远一些，那是因为我站在巨人的肩膀上。”对于用 VC 开发图像处理软件的程序员来讲，OpenCV 就是巨人的肩膀。



### 1.2.1 OpenCV 概述

OpenCV (Open source Computer Vision library) 是 1999 年由 Intel 公司开发的图像处理和计算机视觉开放源码库, 现在由 Willow Garage 实验室提供支持。OpenCV 提供 C++、C 和 Python 接口 (即将支持 Java 接口), 可以运行在 Linux、Windows、Mac OS 和 Android 操作系统上。


OpenCV 拥有包括 500 多个 C/C++ 函数的跨平台的中、高层 API, 具有强大的图像和矩阵运算能力。截止到 2012 年年初, OpenCV 已发展到 2.3.1 版本。每一次版本升级都伴随着众多的函数更新和优化。

早期版本的 OpenCV 包括 CxCore、Cv、CxAux、HighGUI 和 ml 5 个主要模块。CxCore 包括一些基本结构和算法函数, 如数据结构与线性代数支持, 主要提供对各种数据类型的基本运算功能; Cv 主要实现图像处理和计算机视觉功能, 包括图像处理、结构分析、运动分析、物体跟踪、模式识别及摄像机标定等功能; CxAux 是 OpenCV 附加库函数, 包括一些实验性的函数, 如 View Morphing、三维跟踪、PCA (Principal Component Analysis) 和 HMM (Hidden Markov Model) 等函数; HighGUI 是用户交互部分, 包括 GUI (Graphical User Interface, 图形用户接口)、图像视频 I/O 和系统调用函数等; ml 是机器学习模块, 主要内容为分类器。OpenCV 中曾经还有一个 CvCam 模块, 但从 1.1 版本开始 OpenCV 便不再包含它, 其功能被 HighGUI 所取代。

自 2.2 版本之后, OpenCV 将原有的 5 个模块重组为 12 个小模块。

- opencv\_core: 核心功能模块, 包括基本结构、算法、线性代数、离散傅里叶变换、XML (Extensible Markup Language, 扩展标识语言) 和 YAML (YAML Ain't Markup Language, 即“YAML 不是标识语言”之意, YAML 是一种能够被电脑识别的直观的数据序列化格式) 文件 I/O 等。
- opencv\_imgproc: 图像处理模块, 包括滤波、高斯模糊、形态学处理、几何变换、颜色空间转换、直方图计算等。
- opencv\_highgui: 高层用户交互模块, 包括 GUI、图像与视频 I/O 等。
- opencv\_ml: 机器学习模块, 包括支持向量机、决策树、Boosting 方法 (一种用来提高弱分类器准确度的算法) 等。
- opencv\_features2d: 二维特征检测与描述模块, 包括图像特征检测、描述、匹配等。
- opencv\_video: 视频模块, 包括光流法、运动模板、背景减除、目标跟踪等。
- opencv\_objdetect: 目标检测模块, 包括基于 Haar 特征或 LBP (Local Binary Pattern, 局部二值模式) 特征的人脸检测、基于 HOG (Histogram of Oriented Gradient, 方向梯度直方图) 特征的人体检测等。
- opencv\_calib3d: 3D 模块, 包括相机标定、立体匹配、3D 重建等。
- opencv\_flann: FLANN (Fast Library for Approximate Nearest Neighbors, 近似最近邻快速算法库) 接口模块, FLANN 库中包含在高维空间中进行搜索和聚类的算法。

- `opencv_contrib`: 新贡献模块, 包括一些开发者新贡献出来的尚不成熟的代码。
- `opencv_legacy`: 遗留模块, 包括一些过期代码, 用于保持向后兼容。
- `opencv_gpu`: GPU (Graphic Processing Unit, 图形处理器) 加速模块, 包括一些可以利用 CUDA (Compute Unified Device Architecture, 计算统一设备架构是一种由 NVIDIA 公司推出的通用并行计算架构, 即一种编程模型, 它包含了 GPU 内部的并行计算引擎) 进行加速的函数。


 早期版本的 OpenCV 为 Intel 公司的 IPP (Integrated Performance Primitives, 集成高性能原件) 提供了透明接口, 这意味着如果有为 Intel 的特定处理器优化的 IPP 库, OpenCV 将在运行时自动加载这些库。OpenCV 2.0 版的代码已显著优化, 无需 IPP 来提升性能, 因此从 2.0 版开始 OpenCV 不再提供 IPP 接口。

### 1.2.2 在 Visual C++ 中使用 OpenCV

在使用 OpenCV 编程之前, 首先要进行安装和配置。OpenCV 可以从网站 <http://sourceforge.net/projects/opencvlibrary> 上免费下载。安装前要先认真阅读安装指南, 确认 OpenCV 与 VC 版本之间是否匹配, 两者不能任意混搭 (如 OpenCV 2.3.1 不支持 VC 6.0)。OpenCV 中文网站 (<http://www.opencv.org.cn>) 上提供了各种版本的 OpenCV 的中文安装指南, 可以参考。

要先安装 VC, 然后安装 OpenCV。OpenCV 的安装过程非常简单, 按安装向导操作即可, 这里不再赘述。较新版本的 OpenCV 有些情况下直接解压缩即可, 不需要安装, 具体要看相应版本的安装指南。若安装过程中出现 “Add <...>\OpenCV\bin to the system PATH” 选项, 选择该项, 这样可以将 OpenCV 的 “<...>\OpenCV\bin” 路径加入到系统变量 PATH 中。如果是直接解压缩的, 须手动将相应的 “<...>\OpenCV\bin” 的完整路径加入到系统变量 PATH 中。有些函数需要 TBB (Thread Building Blocks, 线程构建模块), 所以需要将 TBB 所在的目录也加入到环境变量 PATH 中。

然后配置 VC, 一是配置全局选项, 指定 OpenCV 的头文件和库文件的路径, 二是配置每个新建项目, 在链接器的附加依赖项中添加所需的库文件 (\*.lib)。各种版本的配置方法是类似的, 但是不同版本的配置参数均不同, 随着版本始终在升级, 此处不再列出, 请读者配置时自行查阅相应版本的安装指南。

 如果需要对 OpenCV 库进行重新编译 (非必需步骤), 可以从 <http://www.cmake.org/cmake/resources/software.html> 下载并安装 CMake 软件, 用 CMake 生成 VC 项目, 然后用 VC 打开项目文件 (针对 VC.NET 版本生成的是 OpenCV.sln), 重新生成即可。

在 VC 中使用 OpenCV, 先创建项目, 然后为项目的 Debug 和 Release 配置附加依赖项 (添加所需的 OpenCV 库文件), 在程序中添加所需的 OpenCV 头文件 (OpenCV 2.2 以后的头文件与以前版本的不同), 即可使用 OpenCV 库中的结构、函数和类, 原型定义及具体用法可到 <http://opencv.itseez.com> 网站上查阅 OpenCV 开发文档或查阅随机安装的开发文档。

OpenCV 中的所有类和函数都放在命名空间 `cv` 中, 为了避免名称冲突, 请在程序中使用的 OpenCV 函数或类名前冠以 “`cv::`”, 如果不想逐个加 “`cv::`”, 只需在程序前加代码 “`using namespace cv`” 即可。

## 1.3 VTK

“横看成岭侧成峰, 远近高低各不同” 是诗人笔下勾勒出的庐山神韵, 利用可视化技术, VC 也能把抽象的数据化成这真山真水般的图景呈现出来, 但这需要 VTK (Visualization Toolkit) 与之 “联袂出演”。

### 1.3.1 VTK 概述

VTK 是一个用于 3D 图形学、图像处理和可视化的跨平台开源函数库, 它是在 OpenGL 的基础上采用面向对象的设计方法发展起来的, 它将可视化开发过程中经常遇到的编程细节屏蔽起来, 封装了图形图像和可视化领域的上百种算法。VTK 包含一个 C++ 类库和 Tcl/Tk、Java、Python 等语言的解释接口层, 可运行于 Linux、Windows、Mac OS 和 UNIX 平台, 支持并行处理。

VTK 是 1993 年作为《*The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*》一书的附件出版的, 该书及其软件是由通用电气公司研发部的 Will Schroeder、Ken Martin 和 Bill Lorensen 用其闲暇时间合作开发的, 由于其开放源码式的授权, 随书一上市后, VTK 很快地建立了其使用及开发者社群。Will Schroeder 和 Ken Martin 于 1998 年离开通用电气公司, 创立了 Kitware 公司。在 Kitware 的支持下, VTK 社群快速地成长, VTK 在学术研究和商业应用领域也倍受重用, 目前已发展到 5.8 版本。


### 1.3.2 在 Visual C++ 中使用 VTK

在使用 VTK 编程之前, 首先要进行安装 (主要是编译 VTK) 和配置。VTK 可以从 <http://www.vtk.org/VTK/resources/software.html> 免费下载, 一般只需下载 VTK 源文件压缩包即可, 如果需要使用样例数据, 还需下载 VTK 数据压缩包。此外, 还要用到 CMake 工具, 可以从 <http://www.cmake.org/cmake/resources/software.html> 免费下载。

先安装好 VC, 再按如下步骤安装 CMake 和 VTK。

- 1) 将 VTK 源代码包解压到 D:\Program Files\VTK 目录下。
- 2) 将 VTK 数据包解压到 D:\Program Files\VTK\VTKData 目录下。

---

 这里的 VTK 目录和 VTKData 目录是开发者自己创建的, 可以自行指定位置, 可以是任何名字, 但是最好不要使用中文命名, 以免出现问题。

---

3) 安装 CMake 工具, 它的安装和 Windows 普通程序安装没有任何区别, 按照程序安装向导一步步安装即可。只有一步需要稍微说明一下, 如图 1-1 所示, 这个对话框是询问是否将 CMake 添加到系统路径, 默认是不添加的, 这主要是为那些习惯命令行操作方式的用户准备的。如果想深入了解 Cmake, 可以选择将其添加到系统路径, 但是即使现在不选择, 以后也可以在 PATH 环境变量中手动添加。

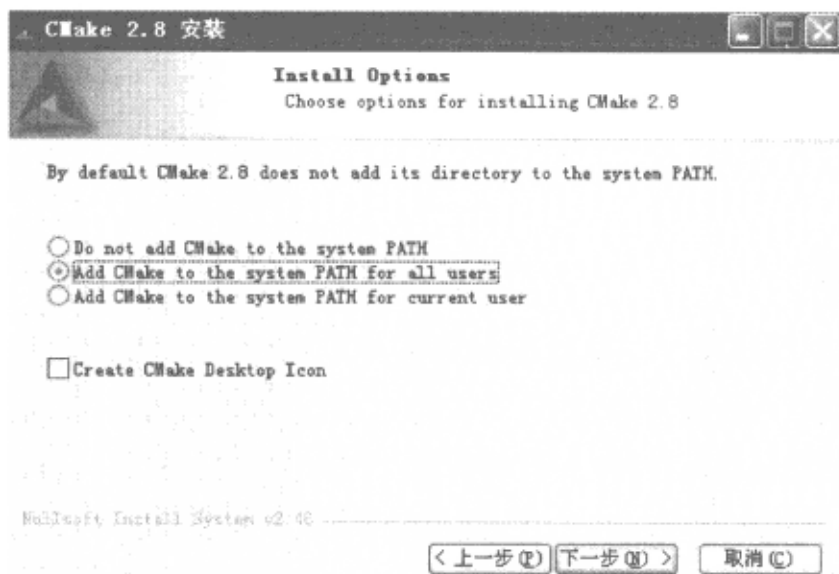


图 1-1 是否将 CMake 添加到系统路径

4) 安装完成后, 打开 CMake, 其主界面如图 1-2 所示。

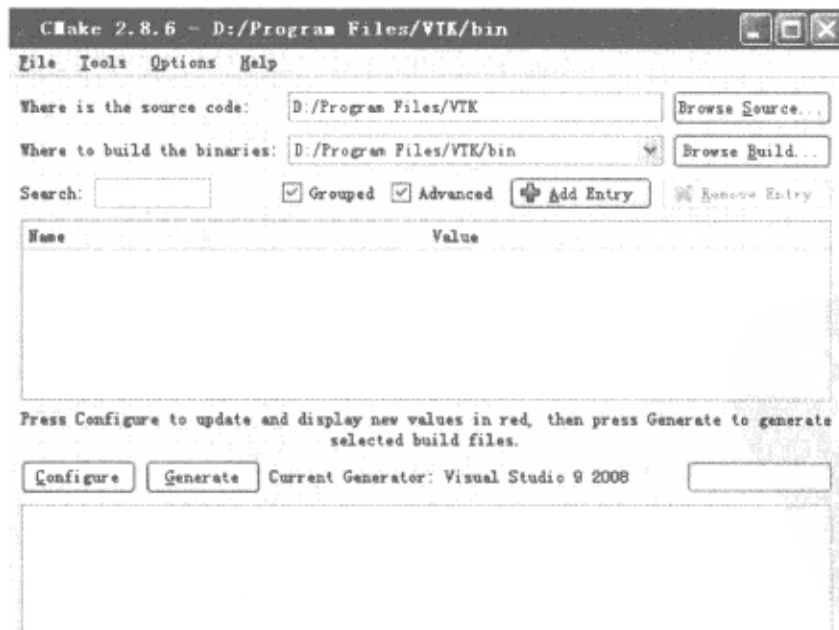


图 1-2 Cmake 主界面

5) 单击按钮 **Browse Source...**, 指定 VTK 源代码所在路径, 这里应该是 D:/Program Files/VTK。

6) 单击按钮 **Browse Build...**，指定将要生成工程的路径，这里选择 D:/Program Files/VTK/bin (bin 是新建的目录)。

📖 CMake 采用的路径写法与 Linux 下相似，采用斜杠 “/”，而不是 Windows 下的反斜杠 “\”，手工输入指定路径时，需特别注意。

7) 单击按钮 **Configure**，打开如图 1-3 所示的编译器选择对话框。

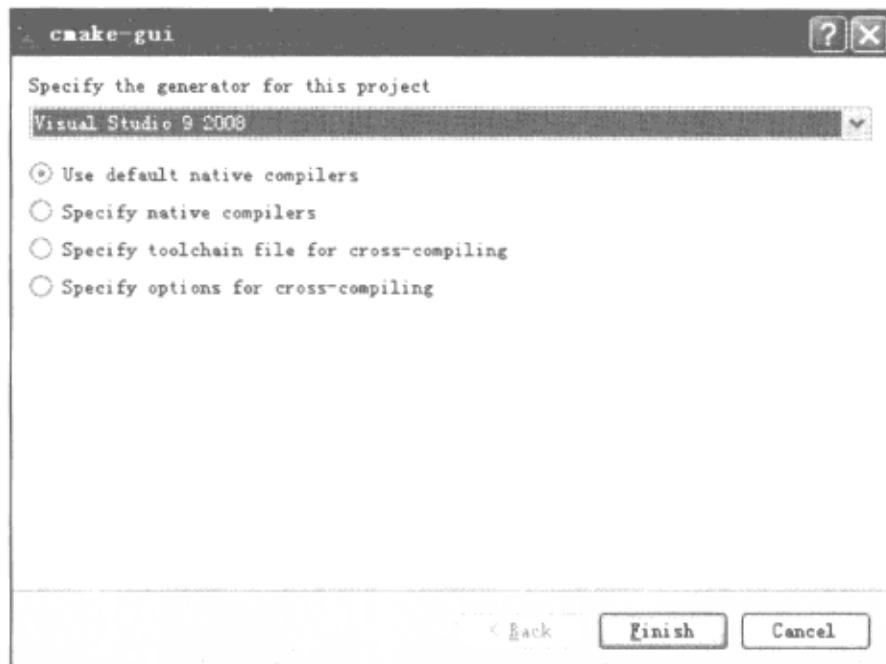


图 1-3 编译器选择对话框

8) 选择 Visual Studio 9 2008 作为编译器，其他选项保持默认设置。

📖 本书中使用的是 VC 2008 开发平台，所以选择 Visual Studio 9 2008 作为编译器，如用其他编译器，则从图 1-3 的下拉列表中选择其他相应的编译器即可。

9) 单击按钮 **Finish** 关闭编译器选择对话框，即可启动配置。

10) 这时按钮 **Configure** 变为按钮 **Stop**，您随时可以中断配置。配置完成后的界面如图 1-4 所示。

📖 如果在图 1-4 中看到 “Could NOT find PythonInterp (missing: PYTHON\_EXECUTABLE)” 信息提示，表明这是与 Python 语言有关的内容，在这不会影响到正常的编译和使用。

11) 根据需要配置 VTK 的生成选项，这里的选项比较多，可以通过在 Search 编辑框中输入选项的名字来快速定位选项，还可以通过它右边的下拉列表选择选项的查看方式，图 1-4 默认是以 Grouped View 来显示。下面介绍几个比较重要的选项。

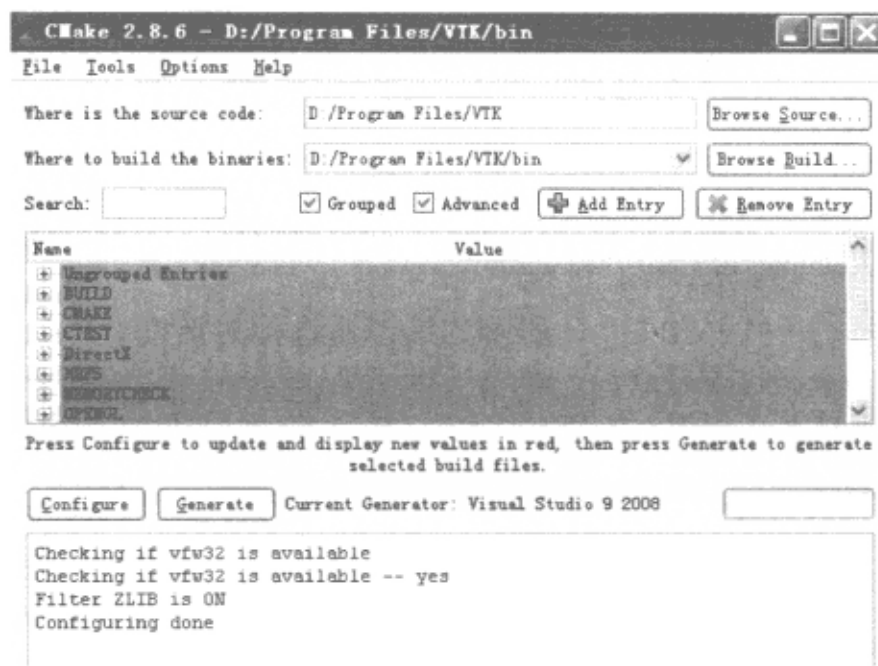


图 1-4 CMake 配置完成后的界面

- VTK\_DATA\_ROOT: 在 VTK 分组里, 用来指明 VTKData 所在目录。
- BUILD\_EXAMPLES: 在 BUILD 分组里, 用来询问是否生成 VTK 所带的例子。

作为初学者, 建议选择生成 VTK 所带的例子, 默认是不选择的。

- BUILD\_SHARED\_LIBS: 在 BUILD 分组里, 用来询问是否生成动态链接库。

如果不选择生成动态链接库, 只生成 LIB 文件, 也可以满足开发的需要。如果选择, 将会生成 DLL 文件, 这些 DLL 文件需要复制到 WINDOWS 的 system32 目录下。如果不选择此项, 应用程序可以独立运行, 但是可执行文件的体积比较大; 如果选择此项, 应用程序必须需要生成的 dll 动态库支持才能运行, 但可执行文件的体积可以做得比较小。建议选择此项, 默认是不选择的。

- VTK\_USE\_GUISUPPORT: 在 VTK 分组里, 如果想在 MFC 应用程序中使用 VTK, 那么就必须选择此项。因为只有打开这个选项, 然后再次单击 **Configure** 按钮才能打开 VTK\_USE\_MFC 选项。

很多人把 VTK 安装好了, 却无法运行 VTK 中与 MFC 混合编程的例子, 提示错误都是缺少 vtkmfc.lib, 问题就出在这个选项, 建议选择此项, 默认是不选择的。

- VTK\_USE\_MFC: 在 VTK 分组里, 如果已打开 VTK\_USE\_GUISUPPORT 选项, 并再次单击按钮 **Configure**, 才能看到这个选项, 建议选择此项, 默认是不选择的。

12) 完成上述配置后, 再次单击按钮 **Configure**, 直到没有红色选项, 如图 1-5 所示。



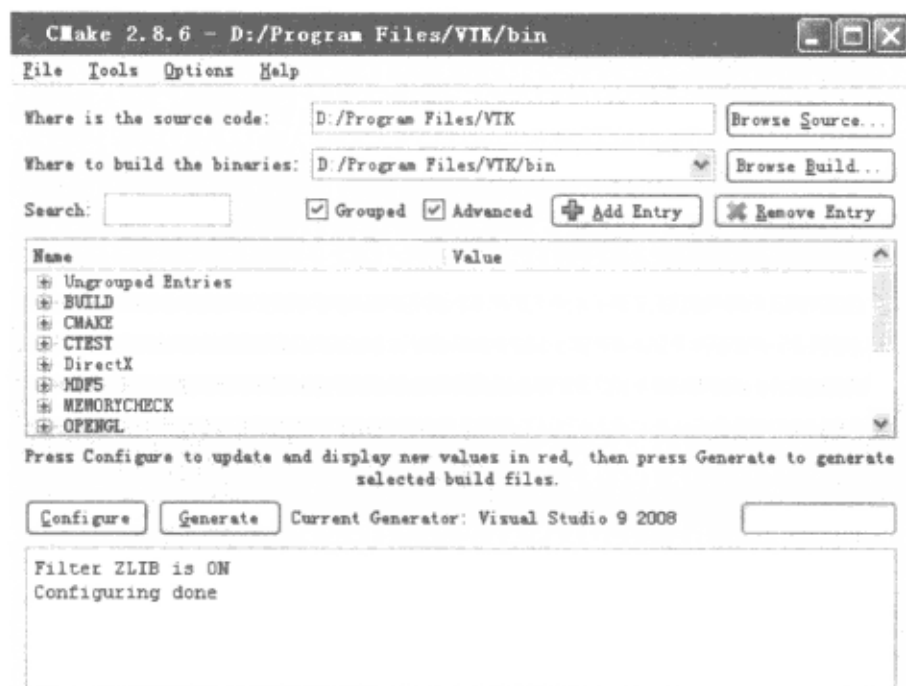


图 1-5 CMake 配置完成时的界面

13) 这时按钮 **Generate** 将变得可用，单击它，生成 VTK 项目。

14) 在 D:\Program Files\VTK\bin 目录下，找到“VTK.sln”解决方案文件，双击这个文件，用 Visual C++ 2008 打开它。

15) 在 VC 2008 开发环境中，选择【生成】菜单下的【批生成】子菜单，打开【批生成】对话框，如图 1-6 所示。

16) 按照图 1-6 所示勾选生成 Debug 版本，单击按钮 **生成(G)**，就开始生成 VTK 项目了。

为了节省编译时间，这里只选择了生成 Debug 版本，对于开发来讲，这就足够了。但是在项目完成，需要部署项目实际应用时，还应该生成并使用 Release 版本。



图 1-6 【批生成】对话框

根据机器的性能不同,生成 VTK 项目一般需要花费十几分钟到几十分钟的时间,生成项目后,将会看到如图 1-7 所示的界面,通过图中输出的信息可以知道,已经成功地生成了 VTK 项目。

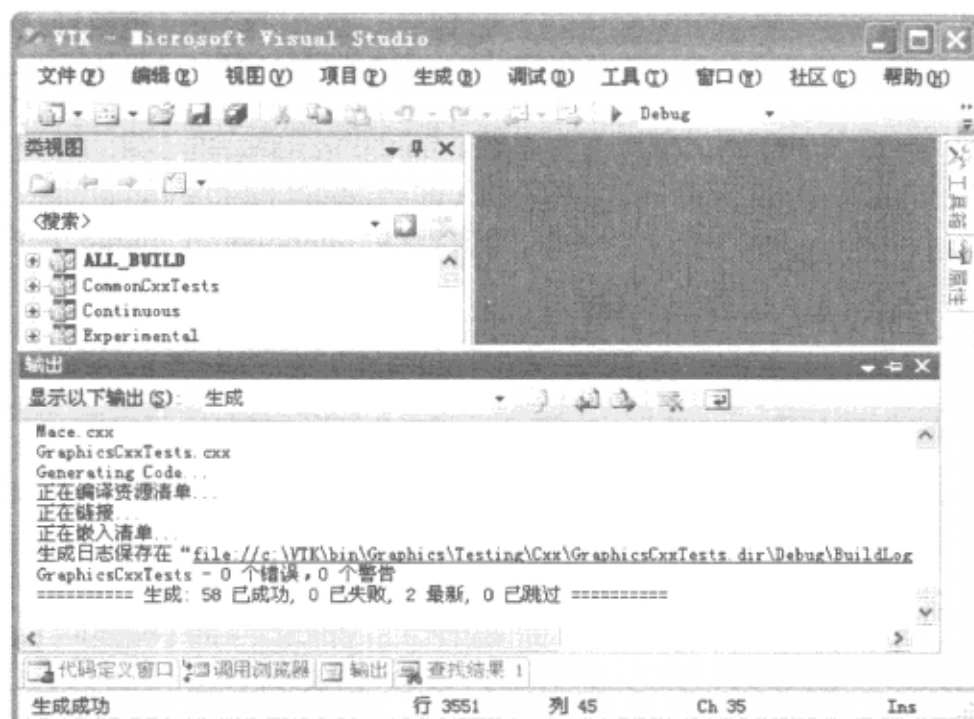



图 1-7 VTK 项目生成成功

安装 VTK 后还不能马上在 VC 中使用它,还需要按如下步骤进行一些配置。

- 1) 打开 D:\Program Files\VTK\bin\bin\debug 目录。
- 2) 以“详细信息”方式查看,按“类型”排序,找到所有以“.dll”为扩展名的文件,将其全部复制到 C:\WINDOWS\system32 目录下。

 这是在 Windows XP 中的目录,如果是其他 Windows 系统,请放到相应的系统目录中,比如对于 Windows 2000 系统,应该放到 C:\WINNT\system32 目录下。这些动态链接库都是使用 VTK 进行项目开发时 VC 2008 要调用的。

- 3) 打开 VC 2008。
- 4) 选择菜单【工具】/【选项】,打开【选项】对话框,并定位到【项目和解决方案】/【VC++ 目录】,如图 1-8 所示。
- 5) 选择【显示以下内容的目录】下拉列表下的【包含文件】选项,然后添加以下目录:
  - D:\Program Files\VTK\COMMON
  - D:\Program Files\VTK\GRAPHICS
  - D:\Program Files\VTK\FILTERING
  - D:\Program Files\VTK\GENERICFILTERING

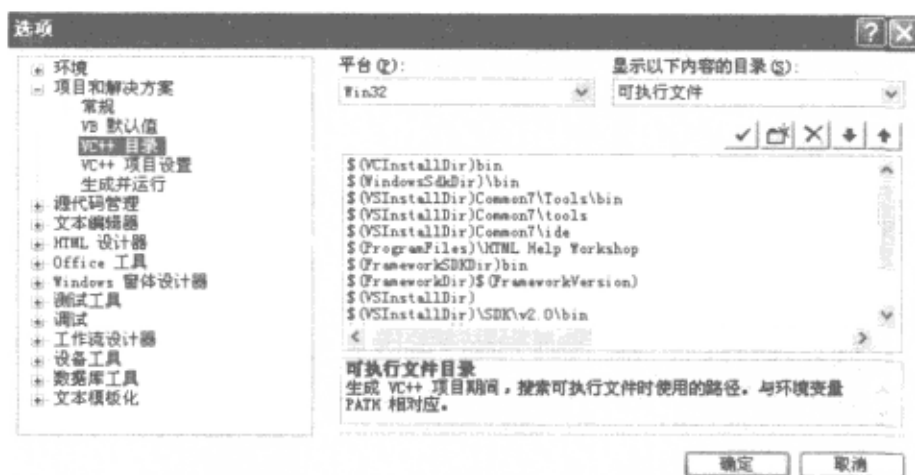



图 1-8 VC 2008 的【选项】对话框

- D:\Program Files\VTK\GEOVIS
- D:\Program Files\VTK\GUISUPPORT\MFC
- D:\Program Files\VTK\HYBRID
- D:\Program Files\VTK\IMAGING
- D:\Program Files\VTK\INFOVIS
- D:\Program Files\VTK\IO
- D:\Program Files\VTK\PARALLEL
- D:\Program Files\VTK\RENDERING
- D:\Program Files\VTK\VIEWS
- D:\Program Files\VTK\VOLUMERENDERING
- D:\Program Files\VTK\WIDGETS
- D:\Program Files\VTK\BIN

6) 选择【显示以下内容的目录】下拉列表下的【库文件】选项，然后添加以下目录：

- D:\Program Files\VTK\BIN\BIN\DEBUG

 上面的配置都是针对 VC 2008 中所有项目进行的全局环境配置，适用于每一个项目，而对于每一个具体的项目来说，还需要进行各自的配置。在 VC 2008 开发环境中，这一般是通过项目属性对话框来进行的，具体做法参见文中后续步骤。

7) 新建或打开一个 VC 2008 项目。

8) 选择菜单【项目】/【属性】，打开项目属性对话框，如图 1-9 所示。

9) 将需要用到的 VTK 库文件，即扩展名为“.lib”的文件，输入到【链接器】/【输入】/【附加依赖项】中，这样 VC 2008 就知道在此项目中具体要使用哪些库文件了。到此为止，配置工作全部完成了。



生成新的数据集输出。通过映射器对象将可视化模型生成的数据集转换到图形模型进行绘制，或者以磁盘文件的形式进行输出。通过演员对象设置几何体的外观属性。“演员”是借用表演行业的术语来表述几何体，演员站在舞台上可以通过灯光、化妆、服装、道具等呈现出不同的造型，几何体在绘制窗口中也可以通过不同的相机、光照、属性设置表现出不同的外观。绘制窗口就相当于这个舞台。通过绘制器对象把几何体、光照及相机视角转换成图像形式的场景，通过绘制窗口在显示设备上生成一个窗口，将绘制器对象加到绘制窗口中便完成了数据的可视化显示过程。绘制窗口支持通过键盘或鼠标的交互。

上述对象在 VTK 中均有定义好的类（详细定义和用法可以下载查阅开发文档或在线查阅 <http://www.vtk.org/VTK/help/documentation.html>），利用这些类创建对象实例，根据项目实际需要构建流水线即可。构建流水线时要注意对象输入和输出之间数据类型的匹配。

## 1.4 经验分享

老话“男怕入错行，女怕嫁错郎”说的是选择的重要性。对程序员来讲，开发平台的选择也非常重要，选择不好会影响到开发效率和软件性能。针对数字图像处理，如果是单纯做算法研究，MATLAB 不失为一个很好的选择。MATLAB 号称科学家的演算纸，专门用于数字图像和计算机视觉处理，开发人员不必把更多的精力放在研究基本图像处理算法的编程实现上，只需专注于新算法的研究和问题求解。如果是单纯做工业视觉软件的开发，Halcon 等软件则是很好的选择，这类工业级开发平台对各种视觉设备的接口做得都很好，而且针对典型的应用问题（如条码识别等）都有成套的解决方案，只要功能明确，可以快速开发出视觉产品原型。如果是做一般的学习和研发，在 Visual C++ 结合 OpenCV 可以作为首选（如果还需要用到可视化，则可以把 VTK 也考虑进来），选择这样的平台，能让开发人员在编程的各个层面上最大限度地施展出自己的才能。

## 第2章 绘图板

“天青色等烟雨，而我在等你，炊烟袅袅升起，隔江千万里……”一曲《青花瓷》用音乐的手法描绘出了烟雨江南水云萌动间伊人初妆的唯美意境。要在计算机中绘制如此美妙的画卷，还需要从最基本的绘图技术开始研究。在数字图像处理应用程序中，也经常需要借助绘图技术来突出显示图像处理结果。本章结合一个绘图板程序案例来介绍绘图中涉及的一些图像和图形学知识，并详细解读通过鼠标交互的方式绘制基本几何图形的方法。

**本章要点：**

- 图形设备接口技术
- 几何图形绘制技术
- 绘图板功能描述
- 绘图板程序总体结构和主要流程
- 绘图板的编程实现

### 2.1 核心技术原理

编写绘图板程序涉及的核心技术主要是图形设备接口技术和几何图形绘制技术。前者主要解决如何把图形输出到图形设备上的问题，后者主要解决各种几何图形如何用程序生成的问题。

#### 2.1.1 图形设备接口技术

图形设备接口常简称为 GDI (Graphics Device Interface)，它的主要任务是负责系统与绘图程序之间的信息交换，处理所有 Windows 程序的图形输出。GDI 是一组 C++ 类，它在驱动程序的协助下把数据描绘在硬件上。在 Windows 操作系统下，绝大多数具备图形界面的应用程序都离不开 GDI，利用 GDI 所提供的众多函数就可以方便地在屏幕、打印机及其他输出设备上输出图形、文本等。GDI 的出现使程序员无须关心硬件设备及设备驱动，就可以将应用程序的输出转化为硬件设备上的输出，实现了程序开发者与硬件设备的分离，大大方便了开发工作（如图 2-1 所示）。



GDI 具有如下特点：

- 1) 不允许程序直接访问物理图形设备硬件，通过称为“设备环境”（Device Context, DC）的抽象接口间接访问硬件。
- 2) 程序需要与图形设备硬件（显示器、打印机等）进行通信时必须首先获得与特定窗口相关联的设备环境。
- 3) 用户无须关心具体的物理设备类型。
- 4) Windows 参考设备环境的数据结构完成数据的输出。



图 2-1 图形设备访问的层次结构

GDI+是 GDI 的下一个版本，它对 GDI 做了很多改进，不仅在 GDI 的基础上添加许多新特性而且对原有的 GDI 功能进行优化。在为开发人员提供的二维矢量图形、文本、图像处理、区域、路径，以及图形数据矩阵等方面构造了一系列相关的类，如 Bitmap（位图类）、Brush（画刷类）、Color（颜色类）、Font（字体类）、Graphics（图形类）、Image（图像类）、Pen（画笔类）和 Region（区域类）等。其中，Graphics 是 GDI+接口中的一个核心类，许多绘图操作都可用它来完成。GDI 的最大优点是屏蔽了数据在设备上绘制的细节，GDI+更好地实现了这个优点。比较而言，GDI 属于中低层 API，需要考虑设备环境，而 GDI+则属于高层 API，不需要考虑设备环境。比如，要设置某个控件的前景和背景色，只需设置其 ForeColor 和 BackColor 属性即可。

### 2.1.2 几何图形绘制技术

现有的很多程序案例都是使用 GDI 处理图形的（本章的案例亦如此），下面首先介绍使用 GDI 绘制几何图形的基本思路，然后再对比着介绍使用 GDI+的绘图方法。

#### 1. 使用 GDI 绘制几何图形

GDI 管理 Windows 应用程序图形的绘制，在应用程序中通过调用 GDI 函数绘制不同尺寸、

颜色、风格的几何图形、文本和位图。MFC 将 GDI 函数封装在一个名为 CDC 的设备环境类中，可以通过调用 CDC 类的成员函数来完成绘图操作。

在使用任何 GDI 绘图函数之前，必须建立一个设备环境 DC。DC 是一个 Windows 数据结构，该结构存储着程序向设备输出时所需要的信息，应用程序利用它定义图形对象及属性，并实现应用程序、设备驱动程序和输出设备之间绘图命令的转换。形象地说，一个设备环境提供了一张画布和一些绘图工具，程序员可以使用不同颜色的工具在其上面绘制点、线、圆和文本。

直接通过 API 函数获取 DC 的方法有两种。

1) 在 WM\_PAINT 消息处理函数中通过调用 API 函数 BeginPaint 获取设备环境，在消息处理函数返回前调用 API 函数 EndPaint 释放设备环境。

2) 在其他函数中通过调用 API 函数 GetDC 获取设备环境，调用 API 函数 ReleaseDC 释放设备环境。

如果采用 MFC 编程，MFC 提供了不同类型的 DC 类，每一个类都封装了 DC 句柄，并且它们的构造函数自动调用获取 DC 的 API 函数，析构函数自动调用释放 DC 的 API 函数。因此，在程序中通过声明一个 MFC 设备环境类的对象就自动获取了一个 DC，而当该对象被销毁时就自动释放了获取的 DC。MFC AppWizard 应用程序向导创建的 OnDraw 函数自动支持所获取的 DC。

MFC 的 DC 类包括 CDC、CPaintDC、CClientDC 和 CWindowDC 等，其中 CDC 类是 MFC 设备环境类的基类，其他 MFC 设备环境类都是 CDC 的派生类。CDC 类既可作为其他 MFC 设备环境类的基类，又可以作为一个一般的设备环境类使用。利用它可以访问设备属性和设置绘图属性。CDC 类对 GDI 的所有绘图函数进行了封装。CPaintDC 类是 OnPaint 函数使用的设备环境类，它代表一个窗口的绘图画面。如果添加 WM\_PAINT 消息处理函数 OnPaint，就需要使用 CPaintDC 类来定义一个设备环境对象。CClientDC 类代表了客户区设备环境。当在客户区实时绘图时，需要利用 CClientDC 类定义一个客户区设备环境。CWindowDC 类代表了整个程序窗口设备环境，可以在整个窗口区域绘图。

画笔和画刷是 Windows 中两种最重要的绘图工具，画笔用于绘制点、线、矩形和椭圆等几何图形，使用画刷可以用指定的颜色和图案来填充绘图区域，这些绘图工具（还有字体、位图和调色板等）统称为 GDI 对象。MFC 对 GDI 对象进行了很好的封装，提供了封装 GDI 对象的类，如 Cpen、CBrush、CFont、Cbitmap 和 CPalette 等，这些类都是 GDI 对象类 CGdiObject 的派生类。

很多涉及颜色的 GDI 函数都需要使用 COLORREF 类型的参数。Windows 用 COLORREF 类型的数据存放颜色，它是一个 32 位整数。任何一种颜色都是由红、绿、蓝 3 种基本颜色组成，COLORREF 类型数据的低位字节存放红色强度值，第 2 个字节存放绿色强度值，第 3 个字节存放蓝色强度值，高位字节为 0，每一种颜色分量的取值范围为 0~255。直接设置 COLORREF 数据不太方便，Windows 提供了 RGB 宏用于设置颜色，RGB(byRed, byGreen, byBlue)将红 (byRed)、绿 (byGreen)、蓝 (byBlue) 分量值转换为 COLORREF 类型的颜色数据。

当用户创建一个用于绘图的设备环境时，该设备环境自动提供了一个宽度为一个像素单位、风格为实黑线 (BLACK\_PEN) 的默认画笔和一个填充色为白色 (WHITE\_BRUSH) 的默认画刷。如果要在设备环境中使用自己的画笔 (画刷) 绘图，首先要创建一个指定风格的画笔 (画刷)，然后将创建的画笔或画刷选入设备环境，最后，在使用该画笔 (画刷) 绘图结束后需要释放该画笔 (画刷)。

创建画笔最简单的方法是调用 CPen 类的一个带参数的构造函数来构造一个 CPen 类画笔对象，以下代码创建了一个红色虚线画笔：

```
CPen PenNew(PS_DASH,1,RGB(255,0,0));
```

创建画笔的第二种方法是首先构造一个没有初始化的 CPen 类画笔对象，然后调用成员函数 CPen::CreatePen 创建定制的画笔工具：

```
CPen PenNew,  
PenNew.CreatePen(PS_DASH,1,RGB(255,0,0));
```

函数 CreatePen 的参数类型与带参数的 CPen 类构造函数完全一样。当画笔对象的声明与创建不在同一个处时 (如需要多次改变画笔)，只能采用这种方法。

画刷有 3 种基本类型：纯色画刷、阴影画刷和图案画刷，CBrush 类提供了多个不同重载形式的构造函数。以下代码创建了 3 种不同类型的画刷：

```
CBrush brush1(RGB(255,0,0));           //创建纯色画刷  
CBrush brush2(HS_DIAGCROSS,RGB(0,255,0)); //创建阴影画刷  
CBrush brush3(&bmp);                   //创建图案画刷
```

创建画刷也可以先构造一个没有初始化的 CBrush 类画刷对象，然后调用 CBrush 类的初始化成员函数创建定制的画刷工具。CBrush 类提供的常用创建函数如下。

- CreateSolidBrush：用指定的颜色创建一个纯色画刷。
- CreateHatchBrush：用指定的阴影样式和颜色创建一个阴影画刷。
- CreatePatternBrush：用位图创建一个图案画刷。
- CreateSysColorBrush：用系统默认颜色创建一个指定阴影样式的画刷。

CDC 类提供了成员函数 CDC::SelectObject 选择用户自己创建的 GDI 对象，该函数有多种重载形式，可以选择用户已定制好的画笔、画刷、字体和位图等不同类型的 GDI 对象。

Windows 预定义了一些简单风格的 GDI 对象，用户使用这些 GDI 对象时，无须自己创建它们，可以直接将它们选入当前的设备环境。这些 GDI 对象称做库存 (Stock) 对象。库存对象包括库存画笔、库存画刷和库存字体等。

创建和选择画笔 (画刷) 工具后，应用程序就可以使用该画笔 (画刷) 绘图了。Windows 中可以绘制的基本几何图形包括点、直线、曲线、矩形、椭圆、弧、扇形、弦形和多边形等。GDI

提供了绘制基本图形的成员函数，这些函数封装在 MFC 的 CDC 类中。绘图函数使用的坐标都是逻辑坐标。

绘图完成后，应该通过调用 CDC 成员函数 CDC::SelectObject 恢复设备环境以前的画笔工具，并通过调用成员函数 CGdiObject::DeleteObject 释放 GDI 对象所占的内存资源。

画笔和画刷对点线的绘制和图形的填充起着很重要的作用，除此之外，还有设备描述表中的绘图模式（又称光栅操作模式）。例如，当绘制一条线段时，该线段的颜色不仅取决于画笔的颜色，而且取决于该线段所在的显示区域的颜色。当使用画笔画线时，它实际上是在画笔像素和目标位置处原像素之间执行一种按位布尔运算，称为“光栅操作”（Raster Operation, ROP）。由于画线操作只涉及两种像素（画笔像素和目标像素），所以这种布尔运算又称为“二元光栅操作”（ROP2）。Windows 定义了 16 种 ROP2 码，用来表示画笔像素和目标像素各种不同的组合方式。可以调用 CDC 的成员函数 SetROP2 改变绘图模式。在用鼠标交互绘制图形时，为了直观地看到所绘制的图形效果，一般采用拖动图形方式。实现拖动绘图，可以用 SetROP2 函数设置绘图模式为 R2\_NOT（表示像素为背景色的取反颜色）。当鼠标移动时，首先把上一个鼠标移动点所决定的图形绘制一次（擦除），再把当前鼠标点决定的图形绘制一次，并记录下鼠标点作为下一个点的移动点，这样当鼠标移动时就实现了拖动效果。

## 2. 使用 GDI+ 绘制几何图形

GDI+ 提供从简单到复杂图形绘制的大量方法，并且可以通过对路径和区域的操作构造出更复杂的图形。

GDI+ 绘图不再使用设备环境或句柄。在使用 GDI 绘图时，必须指定一个设备环境。MFC 为设备环境提供了许多由基类 CDC 派生的设备环境类，如 CPaintDC、CClientDC 和 CWindowDC 等，用来将某个窗口或设备与设备环境类的句柄指针关联起来，所有的绘图操作都与该句柄有关。而 GDI+ 不再使用这个设备环境或句柄，取而代之的是使用 Graphics 对象。与设备环境相类似，Graphics 对象也是将屏幕的某一个窗口与之相关联，并包含绘图操作所需要的相关属性。但是，只有这个 Graphics 对象与设备环境及句柄存在联系，其余的如 Pen、Brush、Image 和 Font 等对象均不再使用设备环境。

GDI+ 绘图方式更简单灵活。GDI 绘图要先创建一个画笔对象，然后将该画笔选入到设备环境中，接下来调用相应的画线函数，最后恢复设备环境中原来的 GDI 对象。而 GDI+ 是先使用 Graphics 类创建一个与设备环境相关联的 Graphics 对象，然后使用 Pen 类进行画笔的创建，最后调用相应的画线方法。Graphics 绘图方法直接将 Pen 对象作为自己的参数，Pen 和设备环境是相互独立的，因而不需要像 GDI 那样恢复设备环境中原来的设置，而且 Pen 和 Graphics 对象的创建不存在先后次序。类似的还有 Brush、Path、Image 和 Font 等。

GDI+ 不再使用“当前位置”。GDI 绘图操作（如画线）中总存在一个叫做“当前位置”的特

殊位置。每次画线都是以此当前位置为起始点，画线操作结束之后，直线的结束点位置又成为了当前位置。设置当前位置的理由是为了提高画线操作的效率，因为在一些情况下，总是一条直线连着另一条直线，首尾相接。有了当前位置的自动更新，就可避免每次画线时都要给出两点的坐标。尽管有其必要性，但是单独绘制一条直线的情况总是比较多的，因此 GDI+取消这个“当前位置”以避免当无法确定“当前位置”时所造成的差错，取而代之的是直接在 DrawLine 中指定直线起止点的坐标。

GDI+形状轮廓绘制和填充采用不同的方法。GDI 总是让形状轮廓绘制和填充使用同一个绘图函数，例如 Rectangle。轮廓绘制需要一个画笔，而填充一个区域需要一个画刷。也就是说，不管是否需要填充所绘制的形状，都需要指定一个画刷，否则 GDI 采用默认的画刷进行填充。这种方式确实带来了许多不便，现在 GDI+将形状轮廓绘制和填充操作分开并采用不同的方法，例如 DrawRectangle 和 FillRectangle 分别用来绘制和填充一个矩形。

GDI+简化了区域的创建方法。GDI 提供了许多区域创建函数，如 CreateRectRgn、CreateEllipticRgn、CreateRoundRectRgn、CreatePolygonRgn 和 CreatePolyPolygonRgn 等。在 GDI+中，为了便于将区域引入矩阵变换操作，GDI+简化一般区域创建的方法，而将更复杂的区域创建交由 Path 接管。由于 Path 对象是与设备环境分离开来的，因而可以直接在 Region 构造函数中加以指定。

GDI+与 GDI 相比，增加了下列新的特性：

1) 渐变画刷。以往 GDI 实现颜色渐变区域的方法是通过使用不同颜色的线条来填充一个裁剪区域。现在 GDI+拓展了 GDI 功能，提供线性渐变和路径渐变画刷来填充一个图形、路径和区域，甚至也可用来绘制直线、曲线等。这里的路径可以视为由各种绘图函数产生的轨迹。

2) 样条曲线。对于曲线而言，最具实际意义的莫过于样条曲线。样条曲线是在生产实践的基础上产生和发展起来的。模线间的设计人员在绘制模线时，先按给定的数据将型值点准确地“点”到图板上。然后，采用一种称为“样条”的工具（一根富有弹性的有机玻璃条或木条），用压铁强迫它通过这些型值点，再适当调整这些压铁，让样条的形态发生变化，直至取得合适的形状，才沿着样条画出所需的曲线。如果把样条看成弹性细梁，那么压铁就可看成作用在这梁上的某些点上的集中力。GDI+的 Graphics::DrawCurve 函数中就有一个这样的参数用来调整集中力的大小。除了样条曲线外，GDI+还支持 GDI 中的 Bezier 曲线。

3) 持久的路径对象。在 GDI 中，路径隶属于一个设备环境，也就是说，一旦设备环境指针超过它的有效期，路径也会被删除。而 GDI+是使用 Graphics 对象来进行绘图操作，并将路径操作从 Graphics 对象分离出来，提供一个 GraphicsPath 类供用户使用。这就是说，不必担心路径对象会受到 Graphics 对象操作的影响，从而可以使用同一个路径对象进行多次的路径绘制操作。

4) 矩阵和矩阵变换。在图形处理过程中常需要对其几何信息进行变换以便产生复杂的新图



形,矩阵是这种图形几何变换最常用的方法。为了满足人们对图形变换的需求,GDI+提供了功能强大的 Matrix 类来实现矩阵的旋转、错切、平移、比例等变换操作。GDI+还支持 Graphics 图形和区域 (Region) 的矩阵变换。

5) Alpha 混色。在图像处理中,Alpha 用来衡量一个像素或图像的透明度。在非压缩的 32 位 RGB 图像中,每个像素由 4 个部分组成:1 个 Alpha 通道和 3 个颜色分量 (R、G 和 B)。当 Alpha 值为 0 时,该像素是完全透明的,而当 Alpha 值为 255 时,则该像素完全不透明。Alpha 混色是将源像素和背景像素的颜色进行混合,最终显示的颜色取决于其 RGB 颜色分量和 Alpha 值。它们之间的关系可用下列公式来表示:显示颜色=源像素颜色  $\times$  Alpha/255+背景颜色  $\times$  (255-Alpha)/255。GDI+的 Color 类定义了 ARGB 颜色数据类型,从而可以通过调整 Alpha 值来改变线条、图像等与背景色混合后的实际效果。

6) 多图片格式的支持。GDI+提供了对各种图片的打开和存储功能。通过 GDI+,能够直接将一个 BMP 文件另存为 JPG 或其他格式的图片文件。

## 2.2 系统功能

“画图”是 Windows 操作系统自带的一款小巧而精美的绘图工具,本章讲解的绘图板模拟实现 Windows 画图程序的部分功能,演示一些基本几何图形的绘制方法。

### 2.2.1 功能描述

- 1) 用鼠标可以绘制的图形包括以下几种:直线、带箭头的线、任意线、圆形、椭圆、矩形。
- 2) 可以对画笔进行设置:画笔的颜色、画笔的线宽、画笔的形状。
- 3) 可以将绘制的所有图形保存到内存中。窗口改变时,自动重绘窗口中的图形。
- 4) 可以将绘制的所有图形保存到文件中,并且可以读取。
- 5) 可以新建空白绘图文档。
- 6) 可以查看最近打开的文档。
- 7) 可以打印用户绘制好的图形,具有打印预览功能。
- 8) 可以撤销和恢复绘图功能。
- 9) 具有图形清空功能。

### 2.2.2 界面效果

绘图板程序界面效果如图 2-2 所示。

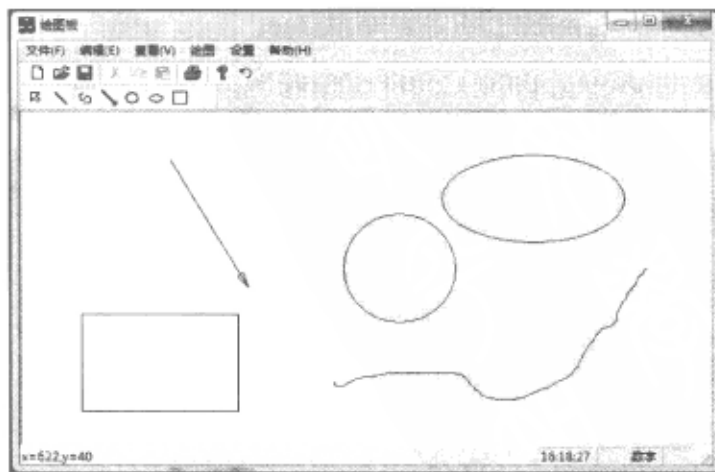


图 2-2 绘图板程序界面效果



## 2.3 系统结构与流程

绘图板采用 MFC 中的单文档“文档/视图”结构。在视图类中完成图形的绘制，在文档类中保存绘制图形的各种数据。

### 2.3.1 总体结构

绘图板的系统结构如图 2-3 所示。

### 2.3.2 主要流程

绘图板程序的主要流程如图 2-4 所示。

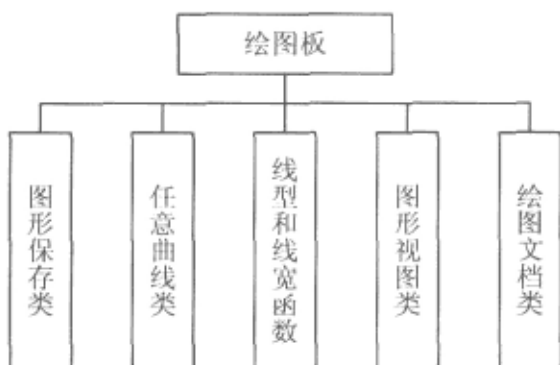


图 2-3 绘图板系统结构图

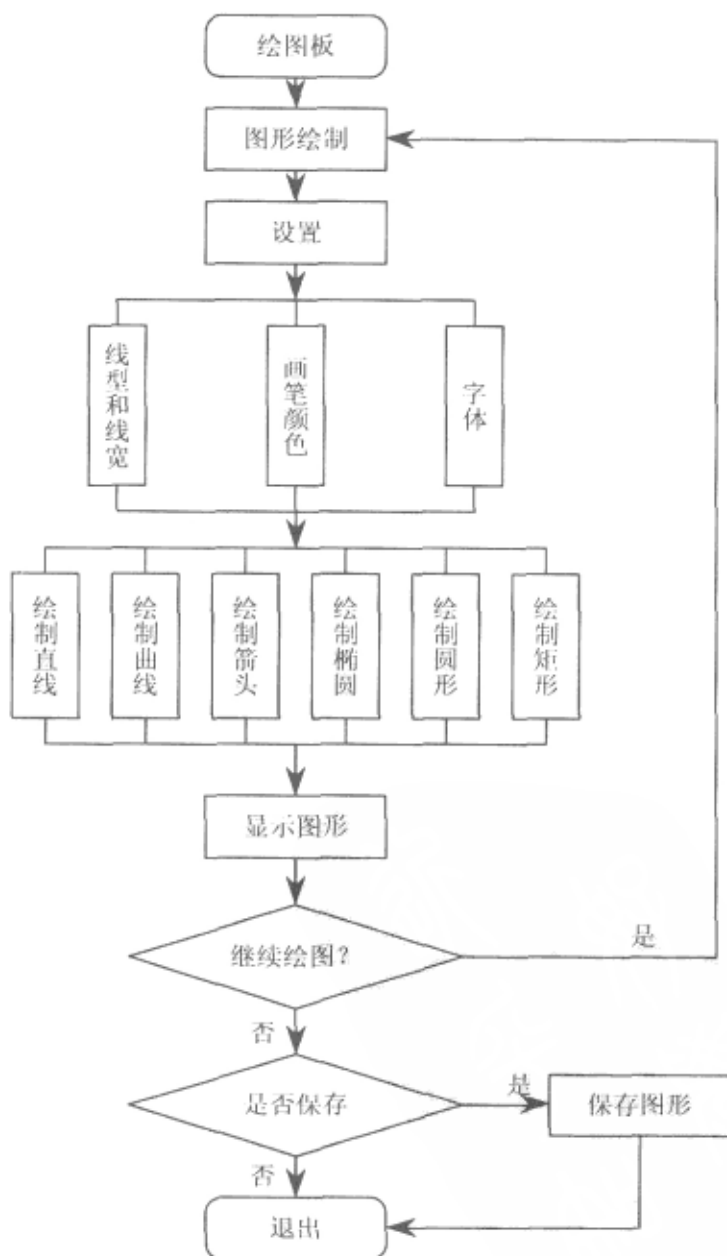


图 2-4 绘图板的流程图

## 2.4 编程实现

绘图板采用 VC 2008 开放平台编程实现，图形接口用的是 GDI。

### 2.4.1 绘图板初始化

绘图板的初始化是在绘图文档类 CMFCDoc 中实现的。该类的主要功能包括：空白文档的创建、画笔和颜色的初始化，以及新文档的打开等。该类的头文件代码如下：

```
#include "Stroke.h"
class CMFCDoc : public CDocument
{
protected:
    CMFCDoc();
    DECLARE_DYNCREATE(CMFCDoc)
public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
public:
    virtual ~CMFCDoc();

    CPen          m_penCur;           // 创建画笔
    CTypedPtrList<CObList, CStroke*> m_strokeList;
    UINT          m_nPenWidth;        // 画笔线宽
    COLORREF      m_color;            // 保存画笔颜色

    CPen*          GetCurrentPen();

    CStroke* NewStroke();
    void InitDocument();
    void DeleteContents();
    BOOL OnOpenDocument(LPCTSTR lpszPathName);
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
protected:
   //{{AFX_MSG(CMFCDoc)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

实现文件具体代码如下：

```
CStroke* CMFCDoc::NewStroke()
```

```

{
    POSITION pos = GetFirstViewPosition();
    CMFCView *pView = (CMFCView*)GetNextView(pos);
    //获取视类的指针
    //获取视类的画笔宽度和颜色
    m_nPenWidth = pView->m_nLineWidth;
    m_color      = pView->m_color;

    CStroke* pStrokeItem = new CStroke(m_nPenWidth,m_color);
    m_strokeList.AddTail(pStrokeItem);
    SetModifiedFlag();

    return pStrokeItem;
}

//-----新建空白文档-----
BOOL CMFCDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)
    //新建空白文档就是把当前的数据全部删除掉
    POSITION pos = GetFirstViewPosition();
    CMFCView *pView = (CMFCView*)GetNextView(pos);
    int max = pView->m_graph.GetSize();
    for(int i=0;i<max;i++)
    {
        pView->m_graph.RemoveAt(max-i-1);
    }
    return TRUE;
}

void CMFCDoc::Serialize(CArchive& ar)    //获取视类的指针
{
    POSITION pos = GetFirstViewPosition();
    CMFCView *pView = (CMFCView*)GetNextView(pos);
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
    m_strokeList.Serialize(ar);
    pView->m_graph.Serialize(ar);
}

```

```

}

//-----初始化画笔和颜色-----
void CMFCDoc::InitDocument()
{
    m_nPenWidth = 0; // default 2 pixel pen width
    m_color = RGB(0,0,0);
    m_penCur.CreatePen(PS_SOLID, m_nPenWidth, m_color);
}

//-----删除保存了任意曲线的数据结构-----
void CMFCDoc::DeleteContents()
{
    while (!m_strokeList.IsEmpty())
    {delete m_strokeList.RemoveHead();
    }
    CDocument::DeleteContents();
}

//-----打开一个新的文档-----
BOOL CMFCDoc::OnOpenDocument(LPCTSTR lpszPathName)
{
    if (!CDocument::OnOpenDocument(lpszPathName))
        return FALSE;
    return TRUE;
}

CPen* CMFCDoc::GetCurrentPen()
{//返回画笔
    return &m_penCur;
}

```

## 2.4.2 图形绘制

绘图功能主要是在视图类 CMFCView 中实现的。该类的主要功能是处理 Windows 消息（主要是来自绘图过程中的人机交互动作）及显示绘制的图形。

传统的消息处理机制是系统把消息放到应用程序消息队列中，应用程序通过 GetMessage 取出消息内容，并通过 DispatchMessage 把消息交给操作系统。而在 MFC 中，不是按照这样的机制进行消息中转处理的。MFC 消息处理采取消息映射的策略实现：在每一个能接收和处理消息的类中，定义一个消息和响应函数的映射表，当接收到消息时，程序从消息表中检索是否存在该消息及其绑定的响应函数，如果存在则将消息交由该响应函数处理。

CMFCView 的头文件中定义了各种变量和消息响应函数，其代码如下：

```

class CMFCView : public CView
{

```

```

protected:
    CMFCView();
    DECLARE_DYNCREATE(CMFCView)

public:
    CMFCDoc* GetDocument();

    DWORD      m_DrawIndex;           //当前所选绘图类型索引
    CPoint      m_random_start;       //任意曲线的起点坐标
    BOOL        m_bDraw;              //判断左键是否按下
    CPoint      m_start;              //起点
    CPoint      m_end;                //终点
    COBArray    m_graph;              //用于保存图形
    COLORREF    m_color;              //保存画笔颜色
    CFont        m_font;              //保存字体
    CString     m_strFontName;        //保存字体名称
    UINTm_nLineWidth;                //保存线宽
    int m_nLineStyle;                //保存线型

    CStroke*    m_pStrokeCur;        //笔画变量
    CPoint      m_ptPrev;             //鼠标响应变量

public:
    float GetDistance(CPoint start, CPoint end); //获得两点坐标距离的函数
    void DrawArrowLine(CPoint start, CPoint end); //画带箭头线的函数
    void DrawLine(CPoint start, CPoint end); //画直线函数
    void DrawEllipse(CPoint start, CPoint end); //画椭圆函数
    void DrawCircle(CPoint start, CPoint& end); //画圆函数
    void DrawRect(CPoint start, CPoint end); //画长方形函数

    // 重载类向导生成的虚函数
    //{AFX_VIRTUAL(CMFCView)
public:
    virtual void OnDraw(CDC* pDC); //重载视图重绘函数
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    //}}AFX_VIRTUAL

public:
    virtual ~CMFCView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;

```

```
#endif

protected:

// 消息映射
protected:
    //{AFX_MSG(CMFCView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point); //鼠标左键按下消息响应函数
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point); //鼠标左键弹起消息响应函数
    afx_msg void OnMouseMove(UINT nFlags, CPoint point); //鼠标移动消息响应函数
    afx_msg void OnRButtonDown(UINT nFlags, CPoint point); //鼠标右键按下消息响应函数

    afx_msg void OnDrawChange(UINT nID); //选择绘图类型消息响应函数
    afx_msg void OnUpdateOperDrawChange(CCmdUI * pCmdUI); //选择绘图类型消息响应函数

    afx_msg void OnEditUndo(); //点击“撤销”消息响应函数
    afx_msg void OnClear(); //点击“清空”消息响应函数
    afx_msg void OnSetLineStyleWidth(); //点击“设置线宽线型”消息响应函数
    afx_msg void OnSetFont(); //点击“设置字体”消息响应函数
    afx_msg void OnSetColor(); //点击“设置颜色”消息响应函数

    DECLARE_MESSAGE_MAP()
};
```

实现文件中主要定义了各种消息响应函数的具体实现。

### 1. 在构造函数中初始化一些变量

在构造函数中添加左键按下标志变量、线型和线宽变量、绘图类型索引变量，并且初始化画笔的颜色。

```
CMFCView::CMFCView() //在构造函数中初始化一些变量
{
    m_bDraw = false; //左键按下标志
    m_nLineStyle = 0; //线型
    m_nLineWidth = 1; //线宽
    m_DrawIndex = 0; //绘图类型索引
    m_color = RGB(0,0,0); //初始化画笔颜色
}
```

### 2. OnDraw 函数

在窗口被遮盖、移动、调整大小后，窗口内的内容都要重绘，对应用程序窗口的客户区进行绘图的所有代码都必须写在 OnDraw 函数中，该函数在窗口重绘时会被自动调用。绘图板程序中，在 OnDraw 函数中实现的主要功能有重绘时显示任意曲线、显示保存在 m\_graph 中的图形、绘制具有橡皮条功能的图形以及图片重绘等功能。具体代码如下：

```
void CMFCView::OnDraw(CDC* pDC)
```



## 数字图像处理典型案例详解

```

{
    CMFCDoc* pDoc = GetDocument();
    //-----重绘时显示任意曲线-----//
    CTypedPtrList<CObList,CStroke*>& strokeList = pDoc->m_strokeList;
    POSITION pos = strokeList.GetHeadPosition();
    while (pos != NULL)
    {
        CStroke* pStroke = strokeList.GetNext(pos);
        pStroke->DrawStroke(pDC);
    }
    //-----重绘时显示保存在 m_graph 中的图形-----//
    int pCount = m_graph.GetSize();
    for(int j=0;j<m_graph.GetSize();j++)
    {
        ((CGraph*)m_graph.GetAt(j))->Draw(pDC);
    }
    //-----绘制有橡皮条功能的图形-----//
    if(m_bDraw == true)
    {
        switch(m_DrawIndex)
        {
            case 1: //绘制直线
                DrawLine(m_start,m_end);
                break;
            case 3: //绘制带箭头直线
                DrawArrowLine(m_start,m_end);
                break;
            case 4: //绘制圆形
                DrawCircle(m_start,m_end);
                break;
            case 5: //绘制椭圆
                DrawEllipse(m_start,m_end);
                break;
            case 6: //绘制矩形
                DrawRect(m_start,m_end);
                break;
        }
    }
    //-----用于文本文件重绘的代码-----//
    if(m_txt == true && m_DrawIndex == 0)
    {
        CRect Rect;
        GetClientRect(&Rect);
        pDC->SelectObject(m_font);
        pDC->DrawText(pBuf,Rect,DT_WORDBREAK);
    }
    //-----用于图片重绘的代码-----//
}

```

```

if(m_bmp == true && m_DrawIndex == 0)
{
    pDC->SetStretchBltMode(COLORONCOLOR);
    StretchDIBits(pDC->GetSafeHdc(), 0, 0, BmpInfo.biWidth,
        BmpInfo.biHeight, 0, 0, BmpInfo.biWidth,
        BmpInfo.biHeight, pBmpData, pBmpInfo, DIB_RGB_COLORS, SRCCOPY);
}
ASSERT_VALID(pDoc);
}

```

### 3. 鼠标左键按下消息响应函数

鼠标是重要的 I/O 设备，图形绘制主要是通过鼠标实现的。鼠标左键按下时，要做的工作主要是设置左键按下标志 `m_bDraw=true`，并记录按下鼠标左键的位置坐标。`m_bDraw` 标志主要是在鼠标移动消息响应函数中强制重绘时用到，在鼠标左键弹起消息响应函数中，`m_bDraw` 会被设置为 `false`。鼠标位置是通过 `point` 参数传递进来的。要绘制任意曲线时，还需要记录构成曲线的点列。代码如下：

```

void CMFCView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 鼠标左键按下消息响应函数
    switch(m_DrawIndex)
    {
        case 1: // 要绘制直线时
            m_bDraw = true;
            m_start = point;
            break;
        case 2: // 要绘制任意曲线时
            {
                m_bDraw = true;
                m_pStrokeCur = GetDocument()->NewStroke();
                m_pStrokeCur->m_pointArray.Add(point);
                m_random_start = point;
            }
            break;
        case 3: // 要绘制带箭头的直线时
            m_bDraw = true;
            m_start = point;
            break;
        case 4: // 要绘制圆形时
            m_bDraw = true;
            m_start = point;
            break;
        case 5: // 要绘制椭圆时
            m_bDraw = true;
            m_start = point;
            break;
    }
}

```

```

        case 6: //要绘制矩形时
            m_bDraw = true;
            m_start = point;
            break;
    }
    m_end = point;

    CView::OnLButtonDown(nFlags, point);
}

```

#### 4. 鼠标左键弹起消息响应函数

图形的最终绘制过程都是在鼠标左键弹起消息响应函数中完成的。该函数中，根据绘图类型索引号 (m\_DrawIndex) 调用相应的成员函数实现绘图，并将图形记录在图形数组 m\_graph 中，以便做撤销处理。具体代码如下：

```

void CMFCView::OnLButtonUp(UINT nFlags, CPoint point)
{ //鼠标左键弹起消息响应函数
    m_bDraw = false;

    CClientDC dc(this);
    CMFCDoc * pDoc = GetDocument();
    CPen* pOldPen = dc.SelectObject(pDoc->GetCurrentPen());
    switch(m_DrawIndex)
    {
        case 1: //绘制直线
        {
            m_end = point;
            DrawLine(m_start, m_end);
            CGraph *graph = new CGraph(m_DrawIndex, m_start, m_end, m_color, m_nLineWidth,
m_nLineStyle);
            m_graph.Add(graph);
        }
        break;
        case 2: //绘制任意曲线
        {
            dc.MoveTo(m_random_start);
            dc.LineTo(point);
            dc.SelectObject(pOldPen);
            m_pStrokeCur->m_pointArray.Add(point);
        }
        break;
        case 3: //绘制带箭头直线
        {
            m_end = point;
            DrawArrowLine(m_start, m_end);
            CGraph *graph = new CGraph(m_DrawIndex, m_start, m_end, m_color, m_nLineWidth,

```

## 第2章

Visual C++

```

m_nLineStyle);
    m_graph.Add(graph);
}
break;
case 4: //绘制圆形
{
    m_end = point;
    DrawCircle(m_start,m_end);
    CGraph *graph = new CGraph(m_DrawIndex, m_start, m_end, m_color, m_nLineWidth,
m_nLineStyle);
    m_graph.Add(graph);
}
break;
case 5: //绘制椭圆
{
    m_end = point;
    DrawEllipse(m_start,m_end);
    CGraph *graph = new CGraph(m_DrawIndex, m_start, m_end, m_color, m_nLineWidth,
m_nLineStyle);
    m_graph.Add(graph);
}
break;
case 6: //绘制矩形
{
    m_end = point;
    DrawRect(m_start,m_end);
    CGraph *graph = new CGraph(m_DrawIndex, m_start, m_end, m_color,
m_nLineWidth, m_nLineStyle);
    m_graph.Add(graph);
}
break;
}
CView::OnLButtonUp(nFlags, point);
}

```

## 5. 鼠标移动消息响应函数

鼠标移动消息响应函数主要做两件事：当鼠标左键未按下时（m\_bDraw==false），检测鼠标位置坐标并显示在状态栏上；当鼠标左键按下时（m\_bDraw==true），通过调用 InvalidateRect 强制重绘窗口。

```

void CMFCView::OnMouseMove(UINT nFlags, CPoint point)
{
    //鼠标移动消息响应函数
    //在状态栏上显示时间
    CString str;
    str.Format("x=%d,y=%d",point.x,point.y);
    //得到框架类窗口的指针 GetParent(), 然后把指针强制转换为框架类的类型(CMainFrame*)
}

```

## 数字图像处理典型案例详解

```
//调用状态栏类的成员函数 setWindowText
((CMainFrame*)GetParent())->m_wndStatusBar.SetWindowText(str);

CClientDC dc(this);
CPen* pOldPen;
CPen pen(m_nLineStyle,m_nLineWidth,m_color);
switch(m_DrawIndex)
{
case 1: //绘制直线
    if(m_bDraw == true)
    {
        m_end = point;
        InvalidateRect(NULL,TRUE);
        break;
    }
case 2: //绘制任意曲线
    if(m_bDraw == true)
    {
        pOldPen= dc.SelectObject(&pen);

        m_pStrokeCur->m_pointArray.Add(point);
        dc.MoveTo(m_random_start);
        dc.LineTo(point);
        m_random_start = point;
    }
    break;
case 3: //绘制带箭头直线
    if(m_bDraw == true)
    {
        m_end = point;
        InvalidateRect(NULL,TRUE);
        break;
    }
    break;
case 4: //绘制圆形
    if(m_bDraw == true)
    {
        m_end = point;
        InvalidateRect(NULL,TRUE);
    }
    break;
case 5: //绘制椭圆
    if(m_bDraw == true)
    {
        m_end = point;
        InvalidateRect(NULL,TRUE);
    }
}
```

```

        break;
    case 6: //绘制矩形
        if(m_bDraw == true)
        {
            m_end = point;
            InvalidateRect(NULL, TRUE);
        }
        break;
    }

    CView::OnMouseMove(nFlags, point);
}

```

## 6. 鼠标右键按下消息响应函数

鼠标右键按下的功能就是弹出菜单，具体实现代码如下：

```

void CMFCView::OnRButtonDown(UINT nFlags, CPoint point)
{ //鼠标右键按下消息响应函数
    ClientToScreen(&point);
    CMenu menu;
    menu.LoadMenu(IDR_MENU1);
    CMenu* pPopup = menu.GetSubMenu(0);

    pPopup->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON, point.x, point.y,
        this); //右键弹出菜单
    CView::OnRButtonDown(nFlags, point);
}

```

## 7. 选择绘图类型消息响应函数

前面代码中提到的绘图类型索引变量 `m_DrawIndex` 就是在这个函数中被赋值的，此函数主要是响应【绘图】菜单消息，根据【绘图】菜单中的子菜单 ID 来设置 `m_DrawIndex`。子菜单 ID 是通过参数 `nID` 传递进来的，`IDM_DRAW_NOTHING` 是【不画图】子菜单的 ID，可以把它理解成绘图类型索引值的基值，`m_DrawIndex` 是 `nID` 与 `IDM_DRAW_NOTHING` 的差值。具体实现代码如下：

```

void CMFCView::OnDrawChange(WORD nID)
{ //选择绘图类型消息响应函数
    m_DrawIndex = nID - IDM_DRAW_NOTHING;
}

```

## 8. 绘制直线

绘制直线（线段）是在 `DrawLine` 函数中通过调用 CDC 的 `MoveTo` 和 `LineTo` 两个成员函数实现的。`MoveTo` 是设置线段起点坐标，`LineTo` 则从当前位置开始绘制一条到指定点的线段。具体实现代码如下。



```

void CMFCView::DrawLine(CPoint start,CPoint end)
{
    //画直线函数
    CClientDC dc(this);
    CPen pen(m_nLineStyle,m_nLineWidth,m_color);
    dc.SelectObject(&pen);
    dc.MoveTo(start);    //设置起点
    dc.LineTo(end);      //绘制线段
}

```

## 9. 绘制带箭头的直线

绘制带箭头的直线是在 DrawArrowLine 中实现的,与画直线类似,不同的是增加了画箭头的操作,箭头是由三个首尾衔接的线段构造出来的,首先设置箭头的方向和角度,然后再确定起止点。具体实现代码如下:

```

void CMFCView::DrawArrowLine(CPoint start, CPoint end)
{
    //画带箭头线的函数
    if(GetDistance(start,end) == 0)
        return;
    float flLength = 15;
    float flAngle = 100;//设置箭头的角度
    float tmpX = (float)(end.x) + ((float)(start.x)-(float)(end.x))*flLength/
    GetDistance(start,end);
    float tmpY = (float)(end.y) + ((float)(start.y)-(float)(end.y))*flLength/
    GetDistance(start,end);
    float fl1X = (tmpX-(float)(end.x))*cos(-flAngle/2) - (tmpY-(float)(end.y))
    *sin(-flAngle/2) + (float)(end.x);
    float fl1Y = (tmpY-(float)(end.y))*cos(-flAngle/2) + (tmpX-(float)(end.x))
    *sin(-flAngle/2) + (float)(end.y);
    float fl2X = (tmpX-(float)(end.x))*cos(flAngle/2) - (tmpY-(float)(end.y))
    *sin(flAngle/2) + (float)(end.x);
    float fl2Y = (tmpY-(float)(end.y))*cos(flAngle/2) + (tmpX-(float)(end.x))
    *sin(flAngle/2) + (float)(end.y);
    CClientDC dc(this);
    CPen pen(m_nLineStyle,m_nLineWidth,m_color);
    dc.SelectObject(&pen);

    dc.MoveTo(start);
    dc.LineTo(end);    //绘制直线

    dc.MoveTo(end.x,end.y);
    dc.LineTo(fl1X,fl1Y);    //绘制箭头线

    dc.MoveTo(end.x,end.y);
    dc.LineTo(fl2X,fl2Y);    //绘制箭头线

    dc.MoveTo(fl1X,fl1Y);
}

```

```
dc.LineTo(fl2X, fl2Y);    //绘制箭头线
}
```

在计算箭头线段的长度时调用了 GetDistance 函数,这是用于计算两点间长度的,按照欧氏距离公式进行计算,代码如下:

```
float CMFCView::GetDistance(CPoint start, CPoint end)
{//得到两点之间的距离
    float flRlt;
    flRlt=(float)((start.x-end.x)*(start.x-end.x)+(start.y-end.y)*(start.y-end.y));
    flRlt=sqrt(flRlt);
    return flRlt;
}
```

## 10. 绘制矩形

绘制矩形是在 DrawRect 函数中通过调用 CDC 的成员函数 Rectangle 实现的,代码如下:

```
void CMFCView::DrawRect(CPoint start, CPoint end)
{
    CClientDC dc(this);
    CPen pen(m_nLineStyle,m_nLineWidth,m_color);
    dc.SelectObject(&pen);
    CBrush *pBrush = CBrush::FromHandle((HBRUSH)GetStockObject(NULL_BRUSH));

    CBrush *pOldBrush = dc.SelectObject(pBrush);
    dc.Rectangle(CRect(m_start,m_end)); //绘制矩形
    dc.SelectObject(pOldBrush);
}
```

## 11. 绘制圆形

绘制圆形是在 DrawCircle 函数中通过调用 CDC 的成员函数 Ellipse(椭圆)实现的,圆其实就是离心率为零的椭圆。Ellipse 需要以椭圆外接矩形的左上角和右下角两点的横、纵坐标为参数,用 Ellipse 画圆时,要重新计算右下角的点坐标,以便将外接矩形调整为正方形。代码如下:

```
void CMFCView::DrawCircle(CPoint start, CPoint& end)
{
    CClientDC dc(this);
    CPen pen(m_nLineStyle,m_nLineWidth,m_color);
    dc.SelectObject(&pen);
    CBrush *pBrush = CBrush::FromHandle((HBRUSH)GetStockObject(NULL_BRUSH));

    int len,lenx,leny;
    lenx = end.x - start.x;
    leny = end.y - start.y;
```

```

        if(leny < lenx)
            len = leny;
        else
            len = lenx;
        //重新计算右下角坐标
        end.x = start.x + len;
        end.y = start.y + len;

        CBrush *pOldBrush = dc.SelectObject(pBrush);
        dc.Ellipse(CRect(m_start,end)); //绘制圆
        dc.SelectObject(pOldBrush);
    }

```

## 12. 绘制椭圆

绘制圆形是在 DrawEllipse 函数中实现的，与绘制圆形一样，也是通过调用 CDC 的成员函数 Ellipse 实现的，代码如下：

```

void CMFCView::DrawEllipse(CPoint start, CPoint end)
{
    CClientDC dc(this);
    CPen pen(m_nLineStyle,m_nLineWidth,m_color);
    dc.SelectObject(&pen);
    CBrush *pBrush = CBrush::FromHandle((HBRUSH)GetStockObject(NULL_BRUSH));
    CBrush *pOldBrush = dc.SelectObject(pBrush);
    dc.Ellipse(CRect(m_start,end)); //绘制椭圆
    dc.SelectObject(pOldBrush);
}

```

## 13. 绘制任意曲线

绘制任意曲线比绘制其他规则几何图形略显复杂，在绘图板中是通过单独定义的类实现的。绘制任意曲线主要用到 CDC 的 MoveTo 和 LineTo 两个成员函数。MoveTo 是设置曲线起点坐标，LineTo 则从当前位置开始绘制一条到指定点的线段。绘制任意曲线，实际上是通过连续绘制折线实现的，这些折线的端点存储在点结构数组 m\_pointArray 中。

在头文件中要做如下定义：

```

class CStroke : public Cobject //绘制任意曲线的类
{
public:
    CStroke(UINT nPenWidth,COLORREF color);

protected:
    CStroke();
    DECLARE_SERIAL(CStroke)

    UINT m_nPenWidth; //保存画笔宽度

```

```

        COLORREF m_color;           //保存画笔颜色
public:
        CArray<CPoint,CPoint> m_pointArray; //记录构成任意曲线的点序列

        BOOL DrawStroke(CDC* pDC);
        virtual void Serialize(CArchive& ar);
};

```

实现代码如下:

```

IMPLEMENT_SERIAL(CStroke, CObject, 2)
CStroke::CStroke()
{
}

CStroke::CStroke(UINT nPenWidth,COLORREF color)
{
    m_nPenWidth = nPenWidth;
    m_color = color;
}

void CStroke::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        ar << (WORD)m_nPenWidth << m_color;
        m_pointArray.Serialize(ar);
    }
    else
    {
        WORD w;
        ar >> w >> m_color;
        m_nPenWidth = w;
        m_pointArray.Serialize(ar);
    }
}

BOOL CStroke::DrawStroke(CDC* pDC)
{
    CPen penStroke;
    if (!penStroke.CreatePen(PS_SOLID, m_nPenWidth, m_color))
        return FALSE;
    CPen* pOldPen = pDC->SelectObject(&penStroke);
    pDC->MoveTo(m_pointArray[0]); //设置曲线起点
    for (int i=1; i < m_pointArray.GetSize(); i++)
    {
        pDC->LineTo(m_pointArray[i]); //连续绘制折线构成任意曲线
    }
}

```

```

    }

    pDC->SelectObject(pOldPen);
    return TRUE;
}

```

#### 14. 设置线型线宽

设置线型线宽在 OnSetLineStyleWidth 函数中通过调用线型线宽设置对话框实现，用 m\_nLineStyle 和 m\_nLineWidth 来保存线型和线宽，具体实现如下：

```

void CMFCView::OnSetLineStyleWidth()
{
    CSettingDlg dlg;
    dlg.m_nLineWidth = m_nLineWidth;
    dlg.m_nLineStyle=m_nLineStyle;
    if(IDOK==dlg.DoModal())
    {
        m_nLineWidth = dlg.m_nLineWidth;
        m_nLineStyle = dlg.m_nLineStyle;
    }
}

```

线型线宽设置对话框 CSettingDlg 的主要功能就是实现线型和线宽的设置，通过变量 m\_nLineStyle 和 m\_nLineWidth 与 OnSetLineStyleWidth 函数传递线型和线宽参数。在头文件中要做如下定义：

```

class CSettingDlg : public Cdialog    //设置线型线宽的对话框类
{
public:
    CSettingDlg(CWnd* pParent = NULL);
    enum { IDD = IDD_DIALOG1 };
    UINT m_nLineWidth;        //线宽
    Int m_nLineStyle;        //线型

protected:
    virtual void DoDataExchange(CDataExchange* pDX);

protected:
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP()
};

```

实现代码如下：

```

CSettingDlg::CSettingDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CSettingDlg::IDD, pParent)
{

```

```

    //{{AFX_DATA_INIT(CSettingDlg)
    m_nLineWidth = 0;
    m_nLineStyle = -1;
    //}}AFX_DATA_INIT
}

void CSettingDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CSettingDlg)
    DDX_Text(pDX, IDC_EDIT2, m_nLineWidth);
    DDX_Radio(pDX, IDC_RADIO1, m_nLineStyle);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CSettingDlg, CDialog)

    ON_WM_PAINT()

END_MESSAGE_MAP()

void CSettingDlg::OnPaint()
{
    CPaintDC dc(this);

    UpdateData();
    CPen pen(m_nLineStyle, m_nLineWidth, m_color);
    dc.SelectObject(&pen);

    CRect rect;                                     //定义矩形区域
    GetDlgItem(IDC_STATIC1)->GetWindowRect(&rect); //得到示例静态空间的矩形区域
    ScreenToClient(&rect);

    dc.MoveTo(rect.left+20, rect.top+rect.Height()/2);
    dc.LineTo(rect.right-20, rect.top+rect.Height()/2);
}

```

## 15. 设置颜色

设置颜色是在 OnSetColor 函数中通过调用 CColorDialog 实现的。CColorDialog 类是用于实现颜色选择的公用对话框。返回的颜色值存储在 m\_color 变量中，代码如下：

```

void CMFCView::OnSetColor()
{
    CColorDialog dlg;
    dlg.m_cc.Flags |= CC_RGBINIT | CC_FULLOPEN;
    dlg.m_cc.rgbResult=m_color;
}

```



## 数字图像处理典型案例详解

```

        if(IDOK==dlg.DoModal())
        {
            m_color=dlg.m_cc.rgbResult;
        }
    }
}

```

### 16. 撤销和恢复绘图操作

MFC 提供了默认的撤销 ID 和恢复 ID, 但是并没有提供默认实现, 本程序的思路是, 定义一个数组和一个数组索引, 每执行一个操作, 就把当前状态存储到数组中, 并把数组索引加 1。撤销时, 把索引减 1 的数组元素恢复到当前文档; 恢复时, 把索引加 1 的数组元素恢复到当前文档。撤销绘图操作函数的实现代码如下:

```

void CMFCView::OnEditUndo()
{
    int num = m_graph.GetSize();
    if(m_graph.GetSize()!=0)
    {
        m_graph.RemoveAt(num-1);
        InvalidateRect(NULL, TRUE);
        num--;
    }
    else
        MessageBox("没有图形供撤销!");
}

```

### 17. 清空图形

清空图形在 OnClear 函数中实现, 该函数的功能就是把画布上的所有图形全部清除, 首先计算画布上图形的数量, 然后用 RemoveAt 方法逐个清除即可, 代码如下:

```

void CMFCView::OnClear()
{
    int max = m_graph.GetSize();
    for(int i=0;i<max;i++)
    {
        m_graph.RemoveAt(max-i-1);
    }
    CMFCDoc * pDoc = GetDocument();
    pDoc->DeleteContents();
    InvalidateRect(NULL, TRUE);
}

```

### 2.4.3 图形保存

图形保存类的主要功能就是保存画笔的颜色、线宽和线型。串行化是微软公司提供的用于对对象进行文件输入/输出的一种机制, 该机制在框架 (Frame) /文档 (Document) /视图 (View)

模式中得到了很好的应用。在保存图形类中，将 CGraph 串行化，可以分以下 5 个步骤实现。

- 1) 派生于 CObject 类。
- 2) 重写 void Serialize( CArchive& ar )。
- 3) 在头文件中定义宏定义 DECLARE\_SERIAL( CGraphPoint )。
- 4) 无参数的构造函数 CGraph()。
- 5) 在实现文件 cpp 中使用宏定义 DECLARE\_SERIAL( CGraph, CObject,1)。

图形保存类要在头文件中做如下定义：

```
class CGraph : public CObject//派生于 CObject 类
{
    DECLARE_SERIAL( CGraph )
public:
    void Serialize( CArchive& ar );
    void Draw(CDC *pDC);          //绘图的方法
    UINT m_DrawIndex;             //绘图的类型
    CPoint m_FirstPoint;          //起始坐标
    CPoint m_SecondPoint;         //结束坐标

    COLORREF m_color;             //保存画笔颜色
    UINT m_nLineWidth;            //保存线宽
    int m_nLineStyle;             //保存线型

    CGraph();
    CGraph(UINT DrawIndex,CPoint FirstPoint,CPoint SecondPoint,COLORREF color,
    UINT LineWidth,int LineStyle); //4 个参数构造函数
    virtual ~CGraph();
};
```

图形保存类的实现代码如下：

```
IMPLEMENT_SERIAL( CGraph, CObject, 1 )
//文档串行化的宏定义 DECLARE_SERIAL( CGraph )

CGraph::CGraph()
{
    // 空构造函数序列化时要用到
}

CGraph::CGraph(UINT DrawIndex,CPoint FirstPoint,CPoint SecondPoint,COLORREF
color,UINT LineWidth,int LineStyle)
{ //5 个参数的构造函数
    this->m_DrawIndex = DrawIndex;
    this->m_FirstPoint = FirstPoint;
    this->m_SecondPoint= SecondPoint;
    this->m_color = color;
```

```

        this->m_nLineStyle = LineStyle;
        this->m_nLineWidth = LineWidth;
    }

CGraph::~CGraph()
{
}

void CGraph::Draw(CDC *pDC)
{
    //CGraph 的绘画方法根据 m_DrawIndex 的不同调用不同的画法
    CPen hpen(m_nLineStyle, m_nLineWidth, m_color);
    CBrush *pBrush = CBrush::FromHandle((HBRUSH)GetStockObject(NULL_BRUSH));
    CBrush *pOldBrush = pDC->SelectObject(pBrush);

    switch(m_DrawIndex)
    {
    case 1:
        pDC->SelectObject(&hpen);
        pDC->MoveTo(m_FirstPoint);
        pDC->LineTo(m_SecondPoint);
        break;
    case 2:
        break;
    case 3:
        {
            float flRlt;
            flRlt=(float)((m_FirstPoint.x-m_SecondPoint.x)*(m_FirstPoint.x-m_SecondPoint.x)+(m_FirstPoint.y-m_SecondPoint.y)*(m_FirstPoint.y-m_SecondPoint.y));
            flRlt=sqrt(flRlt);

            if(flRlt == 0)
                return;
            float flLength = 15;
            float flAngle = 100;//设置箭头的角度

            float tmpX = (float)(m_SecondPoint.x) + ((float)(m_FirstPoint.x)-(float)(m_SecondPoint.x))*flLength/flRlt;
            float tmpY = (float)(m_SecondPoint.y) + ((float)(m_FirstPoint.y)-(float)(m_SecondPoint.y))*flLength/flRlt;

            float fl1X = (tmpX-(float)(m_SecondPoint.x))*cos(-flAngle/2) - (tmpY-(float)(m_SecondPoint.y))*sin(-flAngle/2) + (float)(m_SecondPoint.x);
            float fl1Y = (tmpY-(float)(m_SecondPoint.y))*cos(-flAngle/2) + (tmpX-(float)(m_SecondPoint.x))*sin(-flAngle/2) + (float)(m_SecondPoint.y);

            float fl2X = (tmpX-(float)(m_SecondPoint.x))*cos(flAngle/2) - (tmpY-(float)(m_SecondPoint.y))*sin(flAngle/2) + (float)(m_SecondPoint.x);

```

```

        float fl2Y = (tmpY-(float)(m_SecondPoint.y))*cos(flAngle/2) + (tmpX-
(float)(m_SecondPoint.x))*sin(flAngle/2) + (float)(m_SecondPoint.y);

        pDC->SelectObject(&hpen);
        pDC->MoveTo(m_FirstPoint);
        pDC->LineTo(m_SecondPoint);

        pDC->MoveTo(m_SecondPoint.x,m_SecondPoint.y);
        pDC->LineTo(fl1X,fl1Y);

        pDC->MoveTo(m_SecondPoint.x,m_SecondPoint.y);
        pDC->LineTo(fl2X,fl2Y);

        pDC->MoveTo(fl1X,fl1Y);
        pDC->LineTo(fl2X,fl2Y);
    }
    break;
case 4:
    {
        pDC->SelectObject(&hpen);
        pDC->Ellipse(CRect(m_FirstPoint,m_SecondPoint));
        pDC->SelectObject(pOldBrush);
    }
    break;
case 5:
    {
        pDC->SelectObject(&hpen);
        pDC->Ellipse(CRect(m_FirstPoint,m_SecondPoint));
        pDC->SelectObject(pOldBrush);
    }
    break;
case 6:
    {
        pDC->SelectObject(&hpen);
        pDC->Rectangle(CRect(m_FirstPoint,m_SecondPoint));
        pDC->SelectObject(pOldBrush);
    }
    break;
}
}
)

```

## 2.5 经验分享

1) 参数类型转换。由于 C 系列的语言都是强类型语言，如果类型不匹配的话，需要进行强制类型转换。实际上，Crect 类提供了这样一个成员函数：重载 LPCRECT 操作符，其作用是将

Crect 转换为 LPCRECT 类型。因此,当在程序中给 Rectangle 函数的参数赋值时,如果它发现该参数是一个 Crect 对象,它就会隐式地调用 LPCRECT 操作符,将 Crect 类型的对象转换为 LPRECT 类型。因此,在给函数传递参数时,即使看到的传递的数值类型和所需要的类型不匹配,也不会出现编译和运行的异常。

2) 三维图形的绘制。本章给出的画图板仅实现了部分平面几何图形的绘制功能,如果要实现三维图形的绘制,仅依靠 VC 自身的力量就要颇费一番周折了。一般,做三维图形绘制都要借助专门的开发包与 VC 配合使用,如 OpenGL、VTK 等。

## 第3章 图片浏览器

“借我借我一双慧眼吧，让我把这纷扰看得清清楚楚明明白白真真切切……”

图像承载的是视觉信息，图像显示是几乎所有图像处理程序都要具备的基本功能，在娱乐电子领域，图片浏览更是一项不可或缺的重要功能。本章就来解读图片浏览器“这双慧眼”如何看清图像世界里那“摇曳多姿”的季节。

本章要点：

- 图像文件的编解码技术
- 图像的几何变换技术
- 图像的切换特效技术
- 常见的图像格式分析
- 图片浏览器功能描述
- 图片浏览器的总体结构和主要流程
- 图片浏览器的编程实现

### 3.1 核心技术原理

编写图片浏览器程序涉及的核心技术主要有图像文件的编解码、图像的几何变换和图像的切换特效技术。

#### 3.1.1 图像文件的编解码技术

由于大多数信息是以图像形式存储的，大数据量的图像信息会给存储器的存储容量和通信信道的带宽带来极大的压力，故需要对图像数据进行压缩编码，图像压缩编码从本质上来说就是对要处理的图像数据用一定的规则进行变换和组合，从而达到以尽可能少的代码（符号）来表示尽可能多的数据的目的。压缩是通过编码来实现的，故称之为压缩编码。

图像文件解码即把编码压缩的图像数据还原成原始的表达形式。由于图像编码和图像解码是互逆的过程，所以，只要弄清楚了图像是如何压缩编码的，自然就容易理解如何解码了。压缩编码主要分为无损压缩和有损压缩技术。



## 1. 无损压缩

无损压缩的过程是可逆的，即从压缩后的图像能够完全恢复出原来的图像，信息没有任何丢失。常见的无损压缩编码方法有：Huffman 编码、Shannon-Fano 编码、算术编码、游程编码、线性预测编码和位平面编码。

### (1) Huffman 编码

Huffman 编码的基本原理是：对于出现概率大的信息符号，采用较短的编码，出现概率越小的信息符号，其码长越长，从而达到利用尽可能少的符号来表示源数据，它广泛应用在变长编码方法中。

### (2) Shannon-Fano 编码

Shannon-Fano 编码要符合非续长的条件，在码字中，1 和 0 是独立的，而且差不多是等概率的，这样的准则一方面保证无须用区间来区分码字，同时保证每传送一位码就有 1bit 的信息。值得一提的是，利用该编码，效率最高可达到 100%。

### (3) 算术编码

算术编码的方法是被编码的消息或者符号串表示成 0 和 1 之间的一个间隔，即将其编码成 $[0, 1)$ 之间的浮点小数。符号序列越长，编码表示它的间隔也就越小，表示这一间隔所需的位数也就越多。由于信源的符号序列需要根据某种编码模式生成概率的大小来减少间隔，出现概率大的符号要比出现概率小的符号减少的范围小，因此，只要增加较少的比特就可以对新增加的信息进行编码。

### (4) 游程编码

游程编码 (Running Length Coding)，又称行程编码，是一种利用空间冗余度进行压缩的方法，相对比较简单，也属于统计编码的范畴。其原理非常简单，即将一行中颜色值相同的相邻像素用一个计数值和该颜色值来代替。

### (5) 线性预测编码

预测编码是根据某一模型利用以往的样本值对新样本值进行预测，然后将样本的实际值与其预测值相减得到一个误差值，再对这一误差值进行编码。如果模型足够好且样本序列在时间上相关性较强，那么误差信号的幅度将远远小于原始信号。对差值信号不进行量化而直接编码，称为线性预测编码。线性预测编码的目的是基于通过对每个像素新增的信息进行提取和编码，来消除在空间上较为接近的像素之间的冗余信息，一个像素的新增信息被定义为此像素实际值和预测值之间的差异。

### (6) 位平面编码

对一幅用多个比特表示其灰度值的图像来说，可以将其中的每个比特看成一个二值的平面，也称位面。位平面编码先将多灰度值图像分解成一系列二值图，然后用二元压缩方法对每一幅二

值图进行压缩。位平面编码主要有两个步骤：位平面分解和位平面编码。

## 2. 有损压缩

有损压缩和无损压缩不同，它是以丢失部分信息为代价来换取高压缩比的，其压缩过程是不可逆的，无法完全恢复出原图像，信息有一定的丢失，但它比无损压缩的压缩比高。为了提高压缩比而丢失部分信息造成的失真可以容忍的，许多种有损编码技术有能力根据压缩比率超过100:1的数据，重构出与原图在视觉效果上几乎没有区别的单色图像，并且生成的图像与对原图进行10:1到50:1压缩的图像之间没有本质上的区别。有损压缩方法主要包括有损预测编码方法和变换编码方法。

### (1) 有损预测编码方法

在预测编码中，直接对差值信号进行编码称为线性预测编码，即线性预测编码。与之相对应的是，如果不是直接对差值信号进行编码，而是对差值信号进行量化后再进行编码，就称之为有损预测编码。有损预测编码的方法有很多种，其中，差分脉冲编码调制(DPCM)是一种最具代表性的编码方法。

### (2) 变换编码方法

变换编码不是直接对空域图像信号编码，而是首先将图像数据经过某种正交变换（如傅里叶变换、离散余弦变换、K-L变换等）变换到另一个正交矢量空间（称之为变换域），产生一批变换系数，然后对这些变换系数进行编码处理，从而达到压缩数据的目的。

## 3.1.2 图像的几何变换技术

图像的几何变换也称为空间变换，是指使原始图像按照需要产生大小、形状和位置的变化，属于空域法数字图像处理方式。图像几何变换主要包括图像位置变换和图像尺度变换。

### 1. 图像位置变换

图像位置变换是图像几何变换中的基本变换方法，包括：图像平移、图像旋转、图像镜像和图像转置。

#### (1) 图像平移

图像平移是指将图像中所有像素点按照指定的平移量水平或垂直移动到期望的位置。图像平移只是改变图像在屏幕上的位置，图像本身并不发生变化，其实质是一种坐标的变换，是图像几何变换中最简单的一种。

设源图像中某像素点的原始坐标为 $(x_0, y_0)$ ，向 $x$ 轴方向和 $y$ 轴方向分别平移 $lXOffset$ 和 $lYOffset$ 的距离后，坐标变为 $(x_1, y_1)$ ，则该点移动前后坐标的关系为：

$$\begin{cases} x_1 = x_0 + lXOffset \\ y_1 = y_0 + lYOffset \end{cases} \quad (3-1)$$

利用齐次坐标, 这种关系可以用矩阵变换的方式表示:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & lXOffset \\ 0 & 1 & lYOffset \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-2)$$

由此, 可以计算出平移后每个像素点的新位置, 实现图像的平移。

对变换矩阵求逆, 得到式 (3-2) 的逆变换为:

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -lXOffset \\ 0 & 1 & -lYOffset \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad (3-3)$$

即

$$\begin{cases} x_0 = x_1 - lXOffset \\ y_0 = y_1 - lYOffset \end{cases} \quad (3-4)$$

由此, 可以根据平移后图中的每一像素点的坐标计算出其对应的源图中像素点的坐标, 判断此坐标是否在源图像的范围, 如果超出源图像的范围, 则将该点的像素值统一设置为 255 (白色)。也就是说, 对于平移后不在源图区域的点都处理为不显示。

## (2) 图像旋转

图像旋转是数字图像处理中一种常用的技术, 也是一种比较复杂的图像几何变换。其本质是以图像的中心为原点, 将图像上的所有像素都旋转一个相同的角度。与图像平移一样, 图像旋转也是图像的位置变换, 对于旋转后超出源图像范围的区域同样处理为不显示。

下面具体分析源图像与旋转后图像的对应像素之间的关系, 如图 3-1 所示。

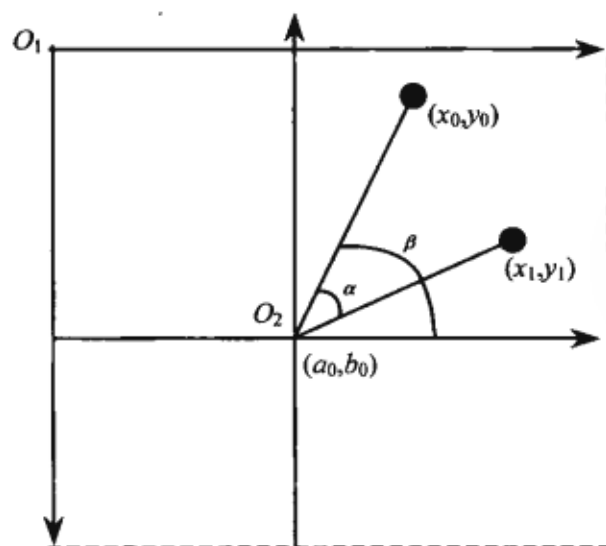


图 3-1 坐标系对照示意图

在笛卡儿坐标系  $O_2$  中, 原始坐标  $(x_0, y_0)$  旋转  $\alpha$  角度后, 坐标变为  $(x_1, y_1)$ , 旋转前

的极坐标表示为:

$$\begin{cases} x_0 = r \cos \beta \\ y_0 = r \sin \beta \end{cases} \quad (3-5)$$

旋转后的极坐标表示为:

$$\begin{cases} x_1 = r \cos(\beta - \alpha) \\ y_1 = r \sin(\beta - \alpha) \end{cases} \quad (3-6)$$

写成矩阵的形式为:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-7)$$

其逆变换为:

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad (3-8)$$

在屏幕坐标系  $O_1$  中, 把笛卡儿中的坐标转换到屏幕坐标系, 计算原始坐标  $(x_0, y_0)$  绕坐标点  $(a_0, b_0)$  旋转  $\alpha$  角后的新坐标  $(x_1, y_1)$ , 可先将笛卡儿坐标系原点  $(0, 0)$  平移到坐标点  $(a_0, b_0)$ , 再根据式 (3-7) 和式 (3-8) 进行旋转, 最后平移回新的坐标原点。

设旋转后新图像的左上角为原点, 旋转前的中心坐标为  $(a_0, b_0)$ , 旋转后的中心坐标为  $(a_1, b_1)$ , 则旋转  $\alpha$  角度后的新坐标  $(x_1, y_1)$ , 可由如下矩阵计算:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a_1 \\ 0 & -1 & b_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \alpha & \sin \alpha & 0 \\ -\sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -a_0 \\ 0 & -1 & b_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-9)$$

其逆变换为:

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \alpha & \sin \alpha & -a_1 \cos \alpha - b_1 \sin \alpha + a \\ \sin \alpha & \cos \alpha & a_1 \sin \alpha - b_1 \cos \alpha + b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad (3-10)$$

即:

$$\begin{cases} x_1 = x_0 \cos \alpha - y_0 \sin \alpha + a_1 \\ y_1 = x_0 \sin \alpha + y_0 \cos \alpha - a_0 \sin \alpha - b_0 \cos \alpha + b_1 \end{cases} \quad (3-11)$$

逆变换为:

$$\begin{cases} x_0 = x_1 \cos \alpha + y_1 \sin \alpha - a_1 \cos \alpha - b_1 \sin \alpha + a_0 \\ y_0 = -x_1 \sin \alpha + y_1 \cos \alpha + a_1 \sin \alpha - b_1 \cos \alpha + b_0 \end{cases} \quad (3-12)$$

由此, 可以根据旋转后图中的每一像素点的坐标计算出其对应的源图中像素点的坐标, 判断

此坐标是否在新计算出的图像范围内, 如果超出范围, 则将该点的像素值处理为不显示, 即统一设置为 255 (白色)。

### (3) 图像镜像

镜像就是成中轴线对称的一种状态。图像的镜像分为两种: 一种是水平镜像, 另一种是垂直镜像。图像的水平镜像操作是将图像左半部分和右半部分以图像垂直中轴线为中心进行镜像对换, 图像的垂直镜像操作是将图像上半部分和下半部分以图像水平中轴线为中心进行镜像对换。

图像镜像的原理比较简单, 若源图像中某一像素点的坐标为  $(x_0, y_0)$ , 镜像后的新坐标为  $(x_1, y_1)$ 。容易得出, 该点关于  $Y$  轴的水平镜像的点的坐标为  $(-x_0, y_0)$ , 该点关于  $X$  轴的垂直镜像的点的坐标为  $(x_0, -y_0)$ 。因此, 在屏幕坐标系中, 设源图像宽度为  $lWidth$ , 高度为  $lHeight$ , 源图像中的点  $(x_0, y_0)$  经过水平镜像后坐标为  $(lWidth - x_0, y_0)$ , 即:

$$\begin{cases} x_1 = lWidth - x_0 \\ y_1 = y_0 \end{cases} \quad (3-13)$$

其矩阵表达式为:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & lWidth \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-14)$$

同理, 对于垂直镜像:

$$\begin{cases} x_1 = x_0 \\ y_1 = lHeight - y_0 \end{cases} \quad (3-15)$$

其矩阵表达式为:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & lHeight \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-16)$$

### (4) 图像转置

图像转置是将图像像素的  $x$  坐标和  $y$  坐标互换。该操作将改变图像的高度和宽度, 转置后图像的高度和宽度将互换。

图像转置也是一种较简单的几何变换, 设源图像中的某个像素点的坐标为  $(x_0, y_0)$ , 其转置后对应新图上的像素点坐标为  $(x_1, y_1)$ , 则二者间关系如下:

$$\begin{cases} x_1 = y_0 \\ y_1 = x_0 \end{cases} \quad (3-17)$$

## 2. 图像尺度变换

图像的尺度变换包括图像缩放及插值算法等技术, 是数字图像几何变换中比较复杂的操作, 不同的算法对最终得到图像的质量影响较大。

## (1) 图像缩放

图像缩放就是根据需要改变图像的大小尺寸,使图像按照一定的比例缩小或放大。图像缩放是一种非常常用的图像处理技术,也是一种相对复杂的图像几何变换技术,因为缩放后产生的新图像的像素,很有可能在源图像中找不到与之相对应的像素点,只能用插值的方法进行近似的处理。通常采用两种插值方法:一种是最近邻插值法,另一种是线性插值法。

设源图像在水平方向和垂直方向的缩放比率分别为  $ZoomX$ 、 $ZoomY$ ,源图像中某个像素点的坐标  $(x_0, y_0)$ ,经缩放后的坐标为  $(x_1, y_1)$ ,则有:

$$\begin{cases} x_1 = x_0 \times ZoomX \\ y_1 = y_0 \times ZoomY \end{cases} \quad (3-18)$$

即:

$$\begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} ZoomX & 0 & 0 \\ 0 & ZoomY & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \quad (3-19)$$

其逆变换为:

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{ZoomX} & 0 & 0 \\ 0 & \frac{1}{ZoomY} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \quad (3-20)$$

所以有:

$$\begin{cases} x_0 = \frac{x_1}{ZoomX} \\ y_0 = \frac{y_1}{ZoomY} \end{cases} \quad (3-21)$$

当  $ZoomX$  和  $ZoomY$  大于 1 时,图像被放大。新图像中的某些像素,对应源图像中的像素可能是实际不存在的。例如在  $ZoomX$  和  $ZoomY$  均为 2 时,新图像中的像素  $(0, 1)$  对应源图像中的像素  $(0, 0.5)$  就是不存在的。此时,只能采取插值法,从源图像中近似地找到或者计算某个像素值并赋给新图像中的对应像素。

当  $ZoomX$  和  $ZoomY$  小于 1 时,图像被缩小,源图像中的某些像素可能被舍弃,例如在  $ZoomX$  和  $ZoomY$  均为 0.5 时,新图像中的像素点只能对应源图像中每行每相邻两个像素取一个,每隔一行取一行。

## (2) 插值算法

数字图像的几何变换,尤其是在进行图像的缩放、旋转变换时,整个变换过程实质由两部分

组成。首先,需要一种算法来完成几何变换本身,用它描述源(输入)图像每个像素如何从其初始位置变换到新(输出)图像对应的位置。其次,还需要一个用于灰度级插值的算法,因为在一般情况下,输入图像的位置坐标 $(x, y)$ 为整数,而变换后输出图像的位置坐标为非整数,灰度值为空,反过来也是如此。因此,在进行图像的几何变换时,除了要进行其本身的几何变换外,还要进行灰度级插值处理。这是由两个独立的算法来完成的。

灰度级插值处理可采用如下两种方法。第一种方法是,把几何变换想象成将输入图像的灰度级像素逐个地转移到输出图像中。如果一个输入像素被映射到4个输出像素之间的位置,则其灰度值就按插值算法在4个输出像素之间进行分配。这种灰度级插值处理称为像素移交法或向前映射法。另一种更有效的灰度级插值处理方法是像素填充(或称为向后映射算法)。在这种算法中,输出像素逐个地映射回原始(输入)图像中,以确定其灰度级。如果一个输出像素被映射到4个输入像素之间,则其灰度值由灰度级插值决定。向后空间变换是向前变换的逆变换。

下面分别介绍几种常用的插值算法。

1) 最近邻插值法。最近邻插值法就是对于通过反向变换得到的一个浮点坐标,对其进行简单的取整,得到一个整数型坐标,这个整数型坐标对应的像素值就是目标像素的像素值。简言之,取浮点坐标最邻近的左上角点(对于DIB是右上角,因为它的扫描行是逆序存储的)对应的像素值。最近邻插值法是一种最简单的插值方法,它思想直观,但得到的图像质量不高。

2) 双线性插值法。双线性插值法就是根据输出图像的宽度和高度,将输入图像的宽度和高度均分,来确定输出图像灰度值的方法。如图3-2所示,假设输出图像的宽度为 $IDstWidth$ ,高度为 $IDstHeight$ ,输入图像的宽度为 $IWidth$ ,高度为 $IHeight$ ,要将输入图像的尺度拉伸或压缩变换至输出图像的宽度方向分为 $IDstWidth$ 等份,高度方向分为 $IDstHeight$ 等份,那么输出图像中任意一点 $(x, y)$ 的灰度值就应该由输入图像中4点 $(a, b)$ 、 $(a+1, b)$ 、 $(a, b+1)$ 和 $(a+1, b+1)$ 的灰度值来确定。

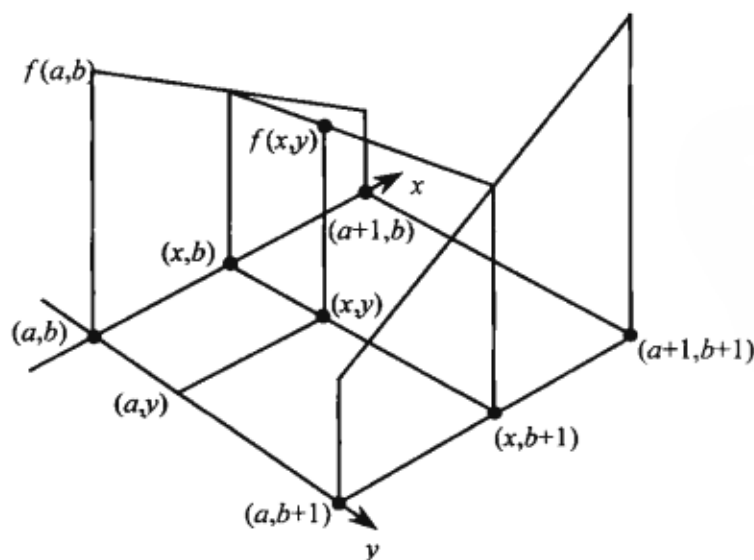


图 3-2 双线性插值示意图



其中,  $a$  和  $b$  的值分别为:

$$a = x \times \frac{lWidth}{lDstWidth}$$

$$b = y \times \frac{lHeight}{lDstHeight}, 0 \leq x < lDstWidth, 0 \leq y < lDstHeight$$

则像素点  $(x, y)$  的灰度值  $f(x, y)$  应为:

$$f(x, y) = (b+1-y)f(x, b) + (y-b)f(x, b+1) \quad (3-22)$$

其中:

$$f(x, b+1) = (x-a)f(a+1, b+1) + (a+1-x)f(a, b+1)$$

$$f(x, b) = (x-a)f(a+1, b) + (a+1-x)f(a, b)$$

以上就是双线性内插值法。值得注意的是: 这种方法缩放后图像质量高, 不会出现像素值不连续的情况, 但是计算量很大, 而且由于双线性插值具有低通滤波器的性质, 使高频分量受损, 所以可能会使图像轮廓在一定程度上变得模糊。

3) 双三次插值法。双三次插值法又叫三次卷积法, 它能够克服以上两种算法的不足, 考虑到一个浮点坐标  $(i+u, j+v)$  周围 16 个邻点, 计算精度高, 计算量大, 目标点的像素值  $f(i+u, j+v)$  可由以下插值公式得到:

$$f(i+u, j+v) = [A][B][C] \quad (3-23)$$

其中:

$$[A] = (S(1+u) \quad S(u) \quad S(1-u) \quad S(2-u))$$

$$[B] = \begin{pmatrix} f(i-1, j-1) & f(i-1, j) & f(i-1, j+1) & f(i-1, j+2) \\ f(i, j-1) & f(i, j) & f(i, j+1) & f(i, j+2) \\ f(i+1, j-1) & f(i+1, j) & f(i+1, j+1) & f(i+1, j+2) \\ f(i+2, j-1) & f(i+2, j) & f(i+2, j+1) & f(i+2, j+2) \end{pmatrix}$$

$$[C] = \begin{bmatrix} S(1+v) \\ S(v) \\ S(1-v) \\ S(2-v) \end{bmatrix}$$

这里,

$$S(\omega) = \begin{cases} 1 - 2|\omega|^2 + |\omega|^3 & |\omega| < 1 \\ 4 - 8|\omega| + 5|\omega|^2 - |\omega|^3 & 1 \leq |\omega| < 2 \\ 0 & |\omega| \geq 2 \end{cases}$$

它是对  $S(\omega) = \frac{\sin(\omega)}{\omega}$  的逼近。

最邻近插值法、双线性插值法、双三次插值法对于旋转变换、错切变换、一般线性变换和非线性变换都适用。

### 3.1.3 图像的切换特效技术

切换特效就是在图像显示出来（从一帧画面切换到下一帧画面）时，不是直接显示整幅图像，而是分块逐步显示，或伴有一些动画效果，从而达到一种“特殊”的显示效果。

切换特效的基本思路就是把图像分成若干小块，按照一定的方向或次序、分阶段地显示或擦除图像块，其中需要解决的问题有：如何划分图像块、如何确定块的操作顺序、如何显示或擦除图像块，以及设置时间延迟等。划分图像块的方法决定了特效显示的方式，图像块操作的次序则决定了图像显示的方向，常见的几种图像特效显示分块方法以及图像操作顺序如表 3-1 所示。

表 3-1 几种图像特效的显示方式

特效名称	图像分块方法	显示方向	图像块操作次序
扫描特效	一行（列）或多行（列）为一块	向下	从上到下
		向上	从下到上
		向右	从左到右
		向左	从右到左
		水平平分	从纵向中央左右
		垂直平分	从横向中央上下
移动特效	一行（列）或多行（列）为一块，操作时，伴随有图像块的位移	向下	从上到下
		向上	从下到上
		向右	从左到右
		向左	从右到左
		水平平分	从纵向中央左右
		垂直平分	从横向中央上下
		水平交叉	横向平分图像，上下部分分别向左右
		垂直交叉	纵向平分图像，左右部分分别向上下
百叶窗特效	等分条状分块	水平方式	同时向上（下）扫描像条
		垂直方式	同时向左（右）扫描像条
栅条特效	等分条状分块	水平方式	同时向上（下）移动各水平图像条，奇偶数图像条的移动方向相反
		垂直方式	同时向左（右）移动各水平图像条，奇偶数图像条的移动方向相反
马赛克特效	将图像分成大小相等的方块		随机显示

### 1. 扫描特效

扫描特效是显示特效中最简单、最基本的一种特效显示方式,其表现是将图像一行一行或一列一列地显示出来或清除掉,很像打开或者收起卷帘的动作,达到一种类似扫描的效果。根据扫描方向的不同,又可分为向下扫描、向上扫描、向左扫描、向右扫描、水平平分和垂直平分6类。

由表3-1可知,扫描特效的基本原理就是把图像的每一行或者每一列分成一个图像块,根据显示方向的不同而选择不同的操作次序。Lena图像向下扫描的效果演示如图3-3所示。



### 2. 移动特效

移动特效通过不断改变图像的大小和位置,使图像呈现水平或者垂直移动的效果。移动显示效果很像关闭和拉开的推拉门,图像从边缘一步一步移动出来或者向边缘移动,直到完全显示或者完全消失。根据移动方向的不同,又可分为向上移动、向下移动、向左移动、向右移动、水平平分、垂直平分、水平交叉和垂直交叉8类。



由表3-1可知,移动特效的基本原理就是将图像的每一行(列)或多行(列)作为一块,操作时伴随有图像块的移动。移动特效采用动态图像分块,在移动中图像块的高度(宽度)是一致的,宽度(高度)会随着时间的改变不断增加,直到等于图像的宽度(高度)为止。图3-4为Lena图像水平平分移动的效果演示。

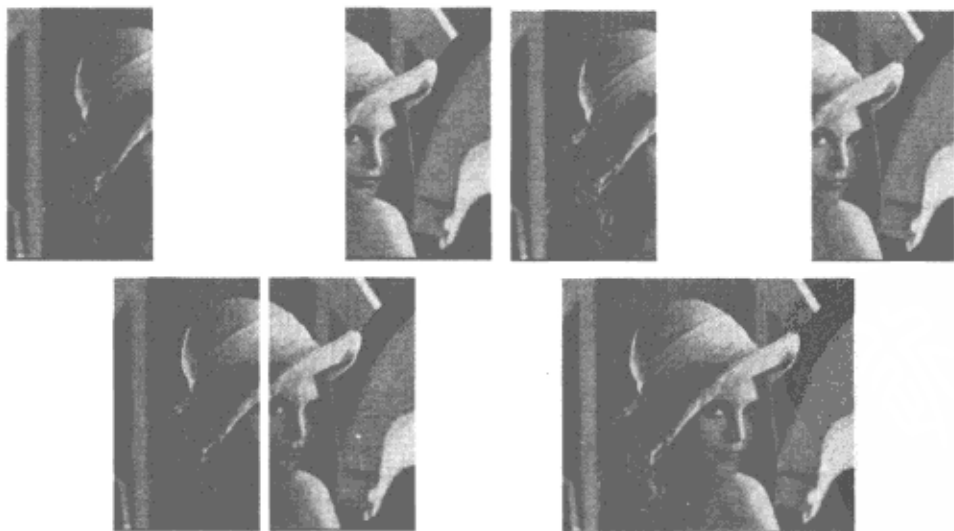


图3-4 Lena图像水平平分移动的效果演示

### 3. 百叶窗特效

百叶窗特效,顾名思义,就是使图像呈现犹如开启或关闭百叶窗那样的效果。在百叶窗显示特效中,图像被分为一行行或者一列列且分别显示出来或者清除掉,显示特效函数将图像分成  $n$

## 数字图像处理典型案例详解

个部分,每部分有  $m$  行或者  $m$  列,显示时同时对所有部分进行扫描。根据显示方式的不同,又可分为垂直百叶窗特效和水平百叶窗特效。

由表 3-1 可知,百叶窗特效的基本原理就是将图像等分为条形块,然后同时分别显示。首先应该确定图像块的高度或者宽度,对于垂直百叶窗而言,图像块的高度一定,宽度可根据具体情况而改变;而对于水平百叶窗而言,图像块的宽度则可以改变。图 3-5 为 Lena 图像垂直百叶窗的效果演示。

#### 4. 栅条特效

栅条特效是移动特效的复杂组合,它是图像分块与移动的混合运用,可分为垂直栅条特效和水平栅条特效两种类型。垂直栅条特效的基本方法是将图像平均分成许多纵向的小条,从左往右数,奇数的图像条向下移动显示,偶数的图像条向上移动显示,或者按相反的方向显示。同理,水平栅条特效显示的基本方法是将图像均分成许多横向的小条,从上往下数,奇数的图像条向左移动显示,偶数的图像条向右移动显示,或者按相反的方向显示。图 3-6 为 Lena 图像垂直栅条的效果演示。

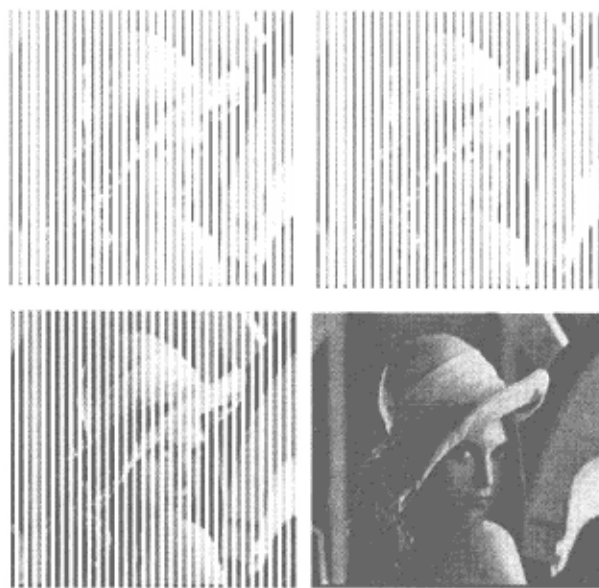


图 3-5 Lena 图像垂直百叶窗的效果演示



图 3-6 Lena 图像垂直栅条的效果演示

#### 5. 马赛克特效

马赛克特效是一种常见的图像处理方法。例如在电视画面中,如果不想显示某个人的身份,就在他的脸部打上马赛克,观众看到的就是很多小方块,从而有效地保护了个人的隐私。从图像处理的角度来看,马赛克特效就是将图像分成大小相等的小方块,然后以随机的形式将所有小方块显示出来,从而使每次马赛克显示的效果不同。

马赛克特效的基本原理很简单,就是将图像分成大小相等的方块,然后进行随机显示。实现

马赛克需要解决两个难点,即如何产生随机的小方块序列和如何将它们显示到相应的位置上。为了解决第一个问题,首先应定义方块的大小,并由此确定小方块的数量;解决第二个问题的关键在于定义一个数组来记录每个小方块的左上角的坐标,显示小方块的位置由该数组来保存。图 3-7 为 Lena 图像马赛克特效的效果演示。



图 3-7 Lena 图像马赛克特效的效果演示

## 3.2 常见的图像格式分析

由于有很多不同类型的图像以及众多不同需求的应用,这就要求图片浏览器能够读出一些比较常见的格式的图像,如 BMP、PCX、TGA、GIF 及 JPEG 等格式,并且能够在这些格式之间相互转换。下面主要分析一下这几种常见的图像格式。

### 3.2.1 BMP 图像

BMP 图像(即位图)是 Windows 系统中最为常见的图像格式。Windows 3.0 以前的版本只支持与设备相关位图 DDB。DDB 是一种内部位图格式,它显示的图像依计算机显示系统的设置不同而不同,因此一般不存储为文件,与通常所说的 BMP 图像不同。在 Windows.h 中,结构体 BITMAP 定义了 DDB 位图的类型、宽度、高度、颜色格式和像素位值等,代码如下:

```
typedef struct tagBITMAP
{
    int bmType;           //位图类型,必须设为 0
    int bmWidth;          //位图宽度
    int bmHeight;         //位图高度
    int bmWidthBytes;     //位图中每一扫描行中的字节数
    BYTE bmPlanes;        //颜色层数
    BYTE bmBitsPixel;     //每一像素所占的位数
}
```

```
void FAR*    bmBits; //存放像素值内存块的地址
}BITMAP;
```

DDB 中不包括颜色信息,显示时是以系统的调色板为基础进行像素的颜色映射的。Windows 只能保证系统调色板的前 20 种颜色稳定不变,所以 DDB 只能保证正确显示少于 20 色的位图。Windows SDK 提供标准的 DDB 位图操作函数;MFC 中定义了 CBitmap 类来说明 DDB 位图,其中封装了与 DDB 位图操作相关的数据结构和函数。

Windows 3.1 以上版本提供了对设备无关位图 DIB 的支持。DIB 位图可以在不同的机器或系统中显示位图所固有的图像。相对于 DDB 而言,DIB 是一种外部位图格式,经常存储为以 BMP 为后缀的位图文件(有时也以 DIB 为后缀)。因此,通常所说的 BMP 图像,即是 DIB 位图。

BMP 位图文件包括四部分,即位图文件头结构 BITMAPFILEHEADER、位图信息头结构 BITMAPINFOHEADER、位图颜色表 RGBQUAD 和位图像素数据。

下面介绍 BMP 文件头、BMP 位图信息头、颜色表这 3 个结构体在 Windows.h 中的定义。

### 1. BMP 文件头

BMP 文件头含有 BMP 文件的类型、文件的大小、位图文件的保留字、位图数据距文件头的偏移量等信息,定义如下:

```
typedef struct tagBITMAPFILEHEADER
{
    UINT bfType;           //位图文件的类型,必须为 BM
    DWORD bfSize;          //位图文件的大小,以字节为单位
    UINT bfReserved1;      //位图文件保留字,必须为 0
    UINT bfReserved2;      //位图文件保留字,必须为 0
    DWORD bfOffBits;       //位图数据距文件头的偏移量,以字节为单位
} BITMAPFILEHEADER;
```

### 2. BMP 位图信息头

BMP 位图信息头用于说明位图的尺寸等信息,定义如下:

```
typedef struct tagBIMAPINFOHEADER
{
    DWORD biSize;          //本结构所占用的字节数
    LONG biWidth;          //位图的宽度,以像素为单位
    LONG biHeight;         //位图的高度,以像素为单位
    WORD biPlanes;         //目标设备的级别,必须为 1
    WORD biBitCount;       //每个像素所需的位数,必须是 1 (双色)、
                          //4 (16 色)、8 (256 色) 或 24 (真彩色) 之一
    DWORD biCompression;   //位图压缩类型,必须是 0 (不压缩)、
                          //1 (BI_RLE8 压缩类型) 或 2 (BI_RLE 压缩类型) 之一
    DWORD biSizeImage;     //位图的大小,以字节为单位
    LONG biXPelsPerMeter;   //位图水平分辨率,每米像素数
    LONG biYPelsPerMeter;   //位图垂直分辨率,每米像素数
```



```

    DWORD biClrUsed;           //位图实际使用的颜色表中的颜色数
    DWORD biClrImportant;      //位图显示过程中重要的颜色数
} BITMAPINFOHEADER;

```

### 3. 颜色表

颜色表用于说明位图中的颜色,它有若干个表项,每一个表项是一个 RGBQUAD 类型的结构,且定义一种颜色,定义如下:

```

typedef struct tagRGBQUAD
{
    BYTE rgbBlue;           //蓝色的亮度 (值范围为 0~255)
    BYTE rgbGreen;          //绿色的亮度 (值范围为 0~255)
    BYTE rgbRed;            //红色的亮度 (值范围为 0~255)
    BYTE rgbReserved;       //保留值,必须为 0
} RGBQUAD;

```

位图信息头和颜色表组成位图信息, BITMAPINFO 结构定义如下:

```

typedef struct tagBITMAPINFO
{
    BITMAPINFOHEADER bmiHeader;
    RGBQUAD bmiColors[i];
} BITMAPINFO;

```

颜色表中 RGBQUAD 结构数据的个数由 *biBitCount* 来确定,当 *biBitCount*=1、4、8 时,分别有 2、16、256 个表项;当 *biBitCount*=24 时,没有颜色表项。

位图数据记录了位图的每一个像素值,记录顺序是在扫描行内从左到右、扫描行之间从下到上。位图的一个像素值所占的字节数如下:

- 当 *biBitCount*=1 时,8 个像素占 1 个字节。
- 当 *biBitCount*=4 时,2 个像素占 1 个字节。
- 当 *biBitCount*=8 时,1 个像素占 1 个字节。
- 当 *biBitCount*=24 时,1 个像素占 3 个字节。

Windows 规定一个扫描行所占的字节数必须是 4 的倍数 (即以 long 为单位),不足的以 0 填充,一个扫描行所占的字节数计算方法如下:

$$DataSizePerLine = (biWidth \times biBitCount + 31) / 8 \quad (3-24)$$

这是程序设计中的一个关键点,无论对图像进行何种变换,都要进行扫描行的 4 字节对齐。

除了上述的 Windows BMP 以外,还有一种与其结构相似,但不完全相同的另一种 BMP 图像,即 OS/2 采用的 BMP。其与 Windows BMP 的主要区别是位图信息结构 (信息头结构和颜色表结构) 不同。而它们的图像位数据的存储方式是完全一样的。本书只讨论 Windows 系统下的 BMP 图像,故对 OS/2 使用的 BMP 不进行详细分析。



### 3.2.2 PCX 图像

PCX 是 ZSoft 公司在开发 PC Paintbrush 软件时设计的一种图像格式，也是在 PC 机上使用时间最久的一种位图格式。PCX 的最新版本支持 24 位色彩（256 色的调色板或全 24 位 RGB），图像的大小可达 64K×64K 像素。PCX 采用游程长度编码来压缩数据，但是有些情况下压缩效率不高，尤其是对于扫描图像或视频图像。PCX 文件格式存在多种版本，其文件头中的“版本号”标识了文件所对应的 PC Paintbrush 版本，不同版本的 PCX 图像特点也有所区别。常见的 PCX 版本号所对应的 PC Paintbrush 版本及图像特点如表 3-2 所示。

表 3-2 不同版本 PCX 图像对应的 PC Paintbrush 版本及图像特点

版本号标识	对应的 PC Paintbrush 版本	图像特点
0	2.5	每像素 1 位
2	2.8	每像素 1 位或 4 位，带调色板信息
3	2.8	每像素 1 位或 4 位，不带调色板（使用默认调色板）
5	3.0 及以后版本	每像素 1 位、4 位、8 位或 24 位

#### 1. 文件结构及文件头

PCX 图像由 3 个部分组成，即文件头、位图数据和一个多达 256 种色彩的调色板。PCX 文件的文件头为固定的 128 字节，其中包含版本号、被打印或扫描图像的分辨率（单位为每英寸点数）、大小（单位为像素数）、每扫描行字节数、每像素位数和彩色平面数等信息。文件还可能包括一个调色板以及表明该调色板是灰度还是彩色的一个代码。文件的核心部分是位图数据。位图数据以类似于 Packbits 压缩法的游程长度压缩形式记录，像素值通常是单字节的索引值，指向调色板中的位置。如果版本号为 5，则文件末尾处还有一个单一的位平面和一个 RGB 值的 256 色调色板，3 种原色各占 1 字节。

PCX 格式图像的文件头结构如表 3-3 所示。

表 3-3 PCX 文件头结构

起始字节	字节数	内 容	解 释
0	1	ZSoft 标志	10 (0x0a)，ZSoft PCX 文件的标志
1	1	版本号	0：PC Paintbrush 2.5 1：PC Paintbrush 2.8，带调色板
2	1	编码	1：PCX 游程长度编码
3	1	位/像素	每个平面的位/像素值，可能值为 1、2、4 或 8
4	8	图像大小	图像边界极限为 Xmin、Ymin、Xmax、Ymax，以像素为单位
12	2	水平分辨率	打印时，X 方向的每英寸点数

(续)

起始字节	字节数	内 容	解 释
14	2	垂直分辨率	打印时，Y 方向的每英寸点数
16	48	文件头调色板	16 色的 EGA/VGA 头调色板
64	1	保留字节	Zsoft 保留，为 0
65	1	平面	彩色/灰度平面数。PCX 图像可以是单彩色，也可以具有多个彩色平面
66	2	每行字节数	每个色彩平面的每行字节数，即存储未压缩图像的一个扫描行所需的字节数，总是偶数
68	2	调色板解释	1：彩色或黑白 2：灰度
70	2	视频屏幕大小 X	视频输出的水平像素数-1
72	2	视频屏幕大小 Y	视频输出的垂直像素数-1
74	54	全空直到文件结束	0

2. PCX 解码

在一个 PCX 文件中可以用到几种不同的记录方法，因此其中必须包含所用方法的标志。在对 PCX 进行解释时，单靠读取版本号是不够的，最可靠的标志是每像素的位数（文件头的第 3 个字节）和色彩平面数（文件头的第 65 个字节），这两个标志与图像色彩数的对应关系如表 3-4 所示。

表 3-4 PCX 数据的解释

每像素的位数	色彩平面数	解 释
1	1	单色
1	2	4 色
1	3	8 色
1	4	16 色
2	1	4 色
2	4	16 色
4	1	16 色
8	1	256 色
8	3	16.7 兆色

色彩平面数说明是否使用了调色板。多于一个色彩平面则没有调色板。如果使用了调色板，则可以由版本号和每像素位数确定 PCX 图像所使用的调色板类型。

3. PCX 图像数据的存储

如果没有使用调色板，则数据是实际的像素值，否则是调色板表项的索引值。当为实际的像素值时，它们按色彩平面和扫描行存储。其存储格式为：

## 数字图像处理典型案例详解

```

第 0 行 RRRRRR...GGGGGG...BBBBBB...
第 1 行 RRRRRR...GGGGGG...BBBBBB...
      :
      :
第 n 行 RRRRRR...GGGGGG...BBBBBB...

```

如果有两个色彩平面，那么色彩是任选的；如果有 3 个色彩平面，其颜色为 RGB；如果有 4 个色彩平面，则颜色信息包含 RGB 和光强。光强位只是给像素一种名义上的较高亮度。

当使用调色板时，数据指调色板的索引值，它们构成一个完整的图像平面，即不会被分解为单独的色彩平面。数据将按如下的简单方式排列（ $i$  是调色板中的索引值）。

```

第 0 行 iiiiiiiiiiiiii...
第 1 行 iiiiiiiiiiiiii...
      :
      :
第 n 行 iiiiiiiiiiiiii...

```

$i$  的长度取决于每像素的位数，如每像素位数为 4，则  $i$  就是半个字节长。

PCX 的编码是以最大 64 个重复单元为一组进行压缩的，不论要记录的是何种类型的数据，都使用同样的游程长度压缩算法。在扫描行中有编码间隔标志，但是，在一个扫描行中的色彩平面之间没有间隔标志。同样，也没有分隔符来标识一个扫描行结束。

#### 4. PCX 图像的调色板

任何 PCX 文件，如果像素位数超过 1 但又只有一个色彩平面，则都需要使用调色板。PCX 图像由 3 种不同的调色板实现。版本代码为 5 的文件最容易确认。如果有一个色彩平面，则它们会在文件结尾处使用 256 色的 VGA 调色板。其他的基于调色板的文件均使用头调色板，而头调色板又有两种可能的实现，即 EGA 和 CGA。三种不同的调色板介绍如下。

##### (1) 位于文件末尾的 256 色 VGA 调色板

256 色的调色板从文件末尾 (EOF) 前 768 个字节开始，而且以十进制码 12 (十六进制 0C) 开始 ( $768=256 \times 1 \text{ 字节} \times 3$ ，每个 R、G 和 B 都是 1 个字节)。因此，值为  $n$  的像素指向调色板中的“EOF-768+3× $n$ ”处，后面 3 个字节分别为该像素红、绿、蓝的值。

##### (2) 16 色的 EGA/VGA 头调色板

头调色板位于第 16~63 字节，共 48 个字节，数据按 3 元组组织，具有 16 组 3 字节数据，每个字节分别对应 R、G 和 B 三原色。对于为 EGA 建立的文件，每种原色只可以有 4 级，所以每个字节提供的 256 个值的范围被分成 4 个区域。每个区域与相应的级相对应：063 对应第 0 级，64~127 对应第 1 级，128~192 对应第 2 级，193~254 对应第 3 级。

##### (3) CGA 调色板

这种调色板现已过时，在 PCX 的版本 5 及更高的版本中不再使用。这种方法只需要字节 16 和字节 19 的最高位数据。

另外，版本 5 或更高版本的 PCX 文件支持 24 位真彩色的 PCX 文件，其色彩平面数为 3。

### 3.2.3 TGA 图像

TGA 格式是 Truevision 公司设计并负责解释的图像格式。TGA 也包含了多种变体，TGA 文件的第三个字节用来区别不是 TGA 的文件。该字节的值（二进制整数）及对应的文件类型如下。

- 0：文件中没有图像数据。
- 1：未压缩的，颜色表图像。
- 2：未压缩的，RGB 图像。
- 3：未压缩的，黑白图像。
- 9：游程长度（Runlength）编码的颜色表图像。
- 10：游程长度（Runlength）编码的 RGB 图像。
- 11：压缩的，黑白图像。
- 32：使用 Huffman、Delta 和 Runlength 编码的颜色表图像。
- 33：使用 Huffman、Delta 和 Runlength 编码的颜色映射图像，二叉树类型处理。

#### 1. 未压缩的颜色表（color-mapped）图像

TGA 未压缩颜色表图像格式的说明如表 3-5 所示。

表 3-5 TGA 未压缩颜色表图像格式说明

偏移（字节）	长度（字节）	描 述
0	1	图像信息字段（见本表的后面）的字符数。本字段是 1 字节无符号整型，指出了图像格式区别字段的长度，其取值范围是 0~255，当它为 0 时，表示没有图像的信息字段
1	1	颜色表的类型。该字段为表示对应格式 1 的图像而包含一个二进制 1
2	1	图像类型码。该字段总为 1，这也是此类型为格式 1 的原因
3	5	颜色表规格，颜色表首址
3	2	颜色表首元表项的整型（低位-高位）索引
5	2	颜色表的长度。颜色表的表项总数，整型（低位-高位）
7	1	颜色表表项的位（bit）数。16 代表 16 位 TGA，24 代表 24 位 TGA，32 代表 32 位 TGA
8	10	图像规格
8	2	图像 X 坐标的起始位置。图像左下角 X 坐标的整型（低位-高位）值
10	2	图像 Y 坐标的起始位置。图像左下角 Y 坐标的整型（低位-高位）值
12	2	图像宽度。以像素为单位，图像宽度的整型（低位-高位）值
14	2	图像高度。以像素为单位，图像高度的整型（低位-高位）值
16	1	图像每像素存储占用位（bit）数

(续)

偏移 (字节)	长度 (字节)	描 述
17	1	图像描述符字节。 1) bits 3~0: 每像素的属性位 (bit) 数。 2) bit 4: 保留, 必须为 0。 3) bit 5: 屏幕起始位置标志。其中, 0 表示原点在左下角, 1 表示原点在左上角, Truevision 图像必须为 0。 4) bits 7~6: 交叉数据存储标志。其中, 00 表示无交叉, 01 表示两路奇/偶交叉, 10 表示四路交叉, 11 表示保留
18	可变	图像信息字段。包含一个自由格式的, 长度是图像记录块偏移 0 处的字节中的值。它常常被忽略 (即偏移 0 处值为 0), 其最大可以包含 255 个字符。如果需要存储更多信息, 可以放在图像数据之后
可变	可变	颜色表数据。起始位置由前一个字段的大小决定, 其长度由单项数据大小和数据项数目决定 (在前面相应的说明字段中给出), 每项是 2 字节、3 字节或 4 字节, 字节中没有使用的位 (bit) 被认为是属性位。 2) 4 字节表项中, 字节 1 表示 BLUE, 字节 2 表示 GREEN, 字节 3 表示 RED, 字节 4 表示属性。 2) 3 字节表项中各字节依次对应 BLUE、GREEN、RED。 3) 2 字节表项中, 两个字节分解成如下形式: arrrrgg gggbbbb, 但是, 由于低位在前, 高位在后的存储顺序, 从文件中读出表项时, 将先读入 gggbbbb 而后读入 arrrrgg, a 表示属性位
可变	可变	图像数据字段。该字段给出了 (高度) × (宽度) 个颜色表项索引, 每个索引以整数个字节的存储形式 (典型的例子如 1 或 2 个字节), 所有的数据都没有符号, 对于 2 字节表项而言, 低位字节是先存储的

2. 未压缩的无颜色表 RGB 图像

TGA 未压缩的无颜色表 RGB 图像格式的说明如表 3-6 所示。

表 3-6 TGA 未压缩的无颜色表 RGB 图像格式说明

偏移 (字节)	长度 (字节)	描 述
0	1	图像信息字段 (见本表的后面) 的字符数。本字段是 1 字节无符号整型, 指出了图像格式区别字段的长度, 其取值范围是 0~255, 当它为 0 时, 表示没有图像的信息字段
1	1	颜色表类型。该字段的内容为 0 或者 1: 0 表示没有颜色表, 1 表示颜色表存在。由于本格式是无颜色表的, 因此此项通常被忽略
2	1	图像类型码。该字段总为 2, 这也是此类型为格式 2 的原因
3	5	颜色表规格。如果颜色表类型字段为 0 则被忽略, 否则用如下 5 个字节描述
3	2	颜色表首址。颜色表首元入口的整型 (低位-高位) 索引

(续)

偏移 (字节)	长度 (字节)	描 述
5	2	颜色表的长度。颜色表的表项总数，整型 (低位-高位)
7	1	颜色表表项的位 (bit) 数。16 代表 16 位 TGA，24 代表 24 位 TGA，32 代表 32 位 TGA
8	10	图像规格
8	2	图像 X 坐标起始位置。图像左下角 X 坐标的整型 (低位-高位) 值
10	2	图像 Y 坐标起始位置。图像左下角 Y 坐标的整型 (低位-高位) 值
12	2	图像宽度。以像素为单位，图像宽度的整型 (低位-高位) 值
14	2	图像高度。以像素为单位，图像高度的整型 (低位-高位) 值
16	1	图像每像素存储占用位 (bit) 数。它的值为 16、24 或 32 等，决定了该图像是 TGA 16，TGA24，TGA 32 等
17	1	<p>图像描述符字节。</p> <p>1) bits 3~0: 每像素对应的属性位的位数。对于 TGA 16，该值为 0 或 1；对于 TGA 24，该值为 0；对于 TGA 32，该值为 8。</p> <p>2) bit 4: 保留，必须为 0。</p> <p>3) bit 5: 屏幕起始位置标志。其中，0 表示原点在左下角，1 表示原点在左上角，Truevision 图像必须为 0。</p> <p>4) bits 7~6: 交叉数据存储标志。其中，00 表示无交叉，01 表示两路奇/偶交叉，10 表示四路交叉，11 表示保留</p>
18	可变	图像信息字段。包含一个自由格式的，长度是图像记录块偏移 0 处的字节中的值。它常常被忽略 (即偏移 0 处值为 0)，注意其最大可以包含 255 个字符。如果需要存储更多信息，可以放在图像数据之后
可变	可变	颜色表数据。如果颜色表类型为 0，则该域不存在，否则越过该域直接读取，图像颜色表规格中描述了每项的字节数，为 2 字节、3 字节或 4 字节
可变	可变	<p>图像数据域。这里存储了 (宽度) × (高度) 个像素，每个像素中的 rgb 色值；该色值包含整数个字节。</p> <p>1) 3 字节表项中各字节依次对应 BLUE、GREEN、RED。</p> <p>2) 2 字节表项中，两个字节分解成如下形式：arrrrgg gggbbbb，但是，由于低位在前，高位在后的存储顺序，从文件中读出表项时，将先读入 gggbbbb 而后读入 arrrrgg；a 表示属性位。</p> <p>3) 4 字节表项包含了分别代表 BLUE、GREEN、RED 及属性的 4 个字节。</p> <p>(由于硬件原因) 有时 TGA 24 类型的图像也像 TGA 32 类型的图像那样存储</p>

### 3. 带颜色表的游程长度 (Runlength) 编码图像

带颜色表的游程长度编码 TGA 图像格式的说明如表 3-7 所示。

表 3-7 带颜色表的游程长度编码 TGA 图像格式说明

偏移 (字节)	长度 (字节)	描 述
0	1	图像信息字段 (见本表的后面) 的字符数。本字段是 1 字节无符号整型, 指出了图像格式区别字段长度, 其取值范围是 0~255, 当它为 0 时, 表示没有图像的信息字段
1	1	颜色表的类型。该字段为表示对应带颜色表的图像而总为 1
2	1	图像类型码。本类型该字段为二进制 9
3	5	颜色表规格。如果颜色表类型字段为 0 则被忽略; 否则用如下 5 个字节描述
3	2	颜色表首址。颜色表首元入口的整型 (低位-高位) 索引
5	2	颜色表的长度。颜色表的表项总数, 整型 (低位-高位)
7	1	颜色表表项的位 (bit) 数。16 代表 16 位 TGA, 24 代表 24 位 TGA, 32 代表 32 位 TGA
8	10	图像规格
8	2	图像 X 坐标起始位置。图像左下角 X 坐标的整型 (低位-高位) 值
10	2	图像 Y 坐标起始位置。图像左下角 Y 坐标的整型 (低位-高位) 值
12	2	图像宽度。以像素为单位, 图像宽度的整型 (低位-高位) 值
14	2	图像高度。以像素为单位, 图像高度的整型 (低位-高位) 值
16	1	图像每像素存储占用位 (bit) 数
17	1	<p>图像描述符字节。</p> <p>1) bits 3~0: 每像素的属性位 (bit) 数。</p> <p>2) bit 4: 保留, 必须为 0。</p> <p>3) bit 5: 屏幕起始位置标志。其中, 0 表示原点在左下角, 1 表示原点在左上角, Truevision 图像必须为 0。</p> <p>4) bits 7~6: 交叉数据存储标志。其中, 00 表示无交叉, 01 表示两路奇/偶交叉, 10 表示四路交叉, 11 表示保留</p>
18	可变	图像信息字段。包含一个自由格式的, 长度是图像记录块偏移 0 处的字节中的值。它常常被忽略 (即偏移 0 处值为 0), 其最大可以包含 255 个字符。如果需要存储更多信息, 可以放在图像数据之后
可变	可变	<p>颜色表数据。起始位置由前一个字段的大小决定; 其长度由单项数据大小和数据项数目决定 (在前面相应的说明字段中给出), 每项是 2 字节、3 字节或 4 字节, 字节中没有使用的位 (bit) 被认为是属性位。</p> <p>1) 4 字节表项中, 字节 1 表示 BLUE, 字节 2 表示 GREEN, 字节 3 表示 RED, 字节 4 表示属性。</p> <p>2) 3 字节表项中各字节依次对应 BLUE、GREEN、RED。</p> <p>3) 2 字节表项中, 两个字节分解成如下形式: arrrrgg gggbbbb, 但是, 由于低位在前, 高位在后的存储顺序, 从文件中读出表项时, 将先读入 gggbbbb 而后读入 arrrrgg, a 表示属性位</p>



(续)

偏移 (字节)	长度 (字节)	描 述
可变	可变	<p>图像数据域。本区域给出了 (宽度) × (高度) 个颜色表索引, 这些索引存放在数据包中, 有两种类型的数据包: run-length 数据包和未加工的数据包。每种类型的数据包含有 1 字节的头信息 (其中指出了数据包类型和数目), 其后是可变长度的数据域, 头信息中最高位为 1 表示 run-length 类型的数据包, 为 0 表示未加工的数据包。</p> <p>对于 run-length 数据包, 头信息含义如下。</p> <p>1 bit id: 7 位的重复记数减 1。由于 7 位表示的最大值为 127, 故最大的运行大小为 128。</p> <p>1: c c c c c c c</p> <p>对于未加工的数据包, 头信息的含义如下。</p> <p>1 bit id: 7 位的像素个数减 1。由于 7 位表示的最大值为 127, 故该类型的一个数据包中像素个数不能大于 128。</p> <p>0: n n n n n n n</p> <p>对于 run-length 数据包而言, 头信息之后是一个简单的颜色索引, 且假定该索引被重复头信息中低 7 位表示的次数, run-length 数据包也许会跨越扫描线 (扫描线起始于某行结束于下一行); 对于未加工的数据包, 头信息之后是颜色索引 (数值由头信息给出), 该类型的数据包也可能跨越扫描线</p>

4. 游程长度 (Runlength) 编码的 RGB TGA 图像

游程长度编码的 RGB TGA 图像格式说明如表 3-8 所示。

表 3-8 游程长度编码的 RGB TGA 图像格式说明

偏移 (字节)	长度 (字节)	描 述
0	1	图像信息字段 (见本表的后面) 的字符数。本字段是 1 字节无符号整型, 指出了图像格式区别字段长度, 其取值范围是 0~255, 当它为 0 时, 表示没有图像的信息字段
1	1	该字段的内容为 0 或者 1, 0 表示没有颜色表, 1 表示颜色表存在。由于本格式是无颜色表的, 因此此项通常被忽略
2	1	图像类型代码。本类型该字段为二进制 10
3	5	颜色表规格。如果颜色表类型字段为 0 则被忽略; 否则描述如下
3	2	颜色表首址。颜色表首元入口的整型 (低位-高位) 索引
5	2	颜色表的长度。颜色表的表项总数, 整型 (低位-高位)
7	1	颜色表表项的位 (bit) 数。16 代表 16 位 TGA、24 代表 24 位 TGA、32 代表 32 位 TGA
8	10	图像规格
8	2	图像 X 坐标起始位置。图像左下角 X 坐标的整型 (低位-高位) 值
10	2	图像 Y 坐标起始位置。图像左下角 Y 坐标的整型 (低位-高位) 值
12	2	图像宽度。以像素为单位, 图像宽度的整型 (低位-高位) 值

(续)

偏移 (字节)	长度 (字节)	描 述
14	2	图像高度。以像素为单位, 图像宽度的整型 (低位~高位) 值
16	1	图像每像素存储占用位 (bit) 数
17	1	<p>图像描述符字节。</p> <p>1) bits 3~0: 每像素的属性位 (bit) 数。</p> <p>2) bit 4: 保留, 必须为 0。</p> <p>3) bit 5: 屏幕起始位置标志。其中, 0 表示原点在左下角, 1 表示原点在左上角, Truevision 图像必须为 0。</p> <p>4) bits 7~6: 交叉数据存储标志。其中, 00 表示无交叉, 01 表示两路奇/偶交叉, 10 表示四路交叉, 11 表示保留</p>
18	可变	图像信息字段。包含一个自由格式的, 长度是图像记录块偏移 0 处的字节中的值。它常常被忽略 (即偏移 0 处值为 0), 其最大可以包含 255 个字符。如果需要存储更多信息, 可以放在图像数据之后
可变	可变	颜色表数据。如果颜色表类型为 0, 则该域不存在, 否则越过该域直接读取; 图像颜色表规格中描述了每项的字节数, 为 2 字节、3 字节或 4 字节
可变	可变	<p>图像数据域。本区域给出了 (宽度) × (高度) 个颜色表索引, 这些索引存放在数据包中, 有两种类型的数据包: run-length 数据包和未加工的数据包, 每种类型的数据包含有 1 字节的头信息 (其中指出了数据包类型和数目), 其后是可变长度的数据域, 头信息中最高位为 1 表示 run-length 类型的数据包, 为 0 表示未加工的数据包。</p> <p>对于 run-length 数据包, 头信息含义如下。</p> <p>1 bit id: 7 位的重复记数减 1。由于 7 位表示的最大值为 127, 故最大的运行大小为 128。</p> <p>1: c      c      c      c      c      c      c</p> <p>对于未加工数据包, 头信息的含义如下。</p> <p>1 bit id: 7 位的像素个数减 1。由于 7 位表示的最大值为 127, 故该类型的一个数据包中像素个数不能大于 128。</p> <p>0: n      n      n      n      n      n      n</p> <p>对于 run length 数据包, 头信息之后是一个简单颜色值, 且假定该值重复的次数为头信息中记录的数, run-length 数据包也许会跨越扫描线 (扫描线起始于某行结束于下一行); 对于未加工的数据包, 头信息之后是颜色值 (数目由头信息指出)。颜色表项自身有 2 字节、3 字节或 4 字节。</p> <p>1) 3 字节表项中各字节依次对应 BLUE、GREEN、RED。</p> <p>2) 2 字节表项中, 两个字节分解成如下形式: arrrrgg gggbbbb, 但是, 由于低位在前, 高位在后的存储顺序, 从文件中读出表项时, 将先读入 gggbbbb 而后读入 arrrrgg; a 表示属性位。</p> <p>3) 4 字节表项包含了分别代表 BLUE、GREEN、RED 及属性的 4 个字节。</p> <p>(由于硬件原因) 有时 TGA 24 类型的图像也像 TGA 32 类型的图像那样存储</p>

### 3.2.4 JPEG 图像

JPEG (Joint Photographic Experts Group, 联合图片专家组) 是由该专家组制定的用于连续色调 (包括灰度和彩色) 静止图像的压缩编码标准。JPEG 标准的压缩编码算法是“多灰度静止图像的数字压缩编码”。

JPEG 标准包括 3 部分, 即编码器、译码器和交换格式。

- 编码器将原始图像的编码压缩成压缩数据。
- 译码器将压缩的图像数据还原成原始图像数据。
- 图像压缩数据以一定的交换格式存储, 格式中包括编码过程中采用的码表等。

JPEG 标准包括以下 4 种运行模式。

- 基本系统 (Baseline System): 基于离散余弦变换 DCT (Discrete Cosine Transform) 进行从左到右、从上到下的顺序扫描编码和重建图像, 实现信息有丢失的图像压缩, 但重建图像的质量要达到难以观察出图像损伤的要求。它采用  $8 \times 8$  像素自适应 DCT 算法量化以及哈夫曼 (Huffman) 型的熵编码器。
- 扩展系统 (Extended System): 选用基于离散余弦变换 DCT 的递增工作方式, 编码过程采用具有自适应能力的算术编码。
- 无失真的预测编码: 采用帧内预测编码及哈夫曼编码 (或算术编码), 可保证重建图像与原始图像完全一样 (即均方误差为零)。
- 分层编码: 以多种分辨率对图像进行编码, 按不同的应用要求可以获得不同分辨率或质量的图像。

JPEG 标准定义了两种基本的压缩算法, 即基于空间线性预测技术差分脉冲码调制 (Differential Pulse Code Modulation, DPCM) 的无失真压缩算法和基于离散余弦 (Discrete Cosine Transform, DCT) 的有失真压缩算法。

JPEG 压缩标准的压缩比是通过量化因子 ( $Q$  因子) 来控制的。 $Q$  因子用来确定原始图像的采样精度, 并产生一个 JPEG 量化矩阵, 即:

$$QM[i,j] = \frac{Q}{50} v[i,j] \quad (3-25)$$

式中  $QM[i,j]$  为量化矩阵,  $Q$  是量化因子,  $v[i,j]$  是图像默认清晰度表。

量化矩阵用来量化 DCT 变换产生的频率系数, 量化后的系数值减少, 0 值的数目大大增加。 $Q$  因子越大, 量化后的 0 值越多, 压缩比越大, 因此  $Q$  因子决定着 JPEG 的压缩比。JPEG 的无失真压缩率为 4:1, 有失真压缩率为 10:1 ~ 100:1。在压缩率小于 40:1 时, 人眼基本上分辨不出图像的效果变化, 可认为是“主观无失真压缩”。

### 3.2.5 GIF 图像

GIF (Graphics Interchange Format, 图形交换格式) 文件由 CompuServe 公司开发并持有该图形文件格式的版权。GIF 图像是基于颜色表存储的, 即图像中每一点的存储数据是该点的颜色对应于颜色列表 (即调色板) 的索引值。GIF 图像最多只支持 8 位存储位, 即最多支持 256 色图像。GIF 文件内部分成许多存储块, 用来存储多幅图像或者是决定图像表现行为的控制块, 可用于实现动画和交互式应用。GIF 文件使用 LZW 压缩算法压缩图像。

GIF 文件内部是按块划分的, 包括控制块 (Control Block) 和数据块 (Data Sub-blocks) 两种:

- 控制块控制数据块的行为, 不同的控制块包含一些不同的控制参数。
- 数据块只包含一些 8-bit 的字符流, 由它前面的控制块来决定它的功能。每个数据块大小从 0~255 个字节不等。数据块的第一个字节存储这个数据块大小 (字节数), 但数据块的大小不包括这个字节。所以即使空的数据块也有一个字节, 即数据块的大小为  $0 \times 00$ 。

一个 GIF 数据块的结构如图 3-8 所示。

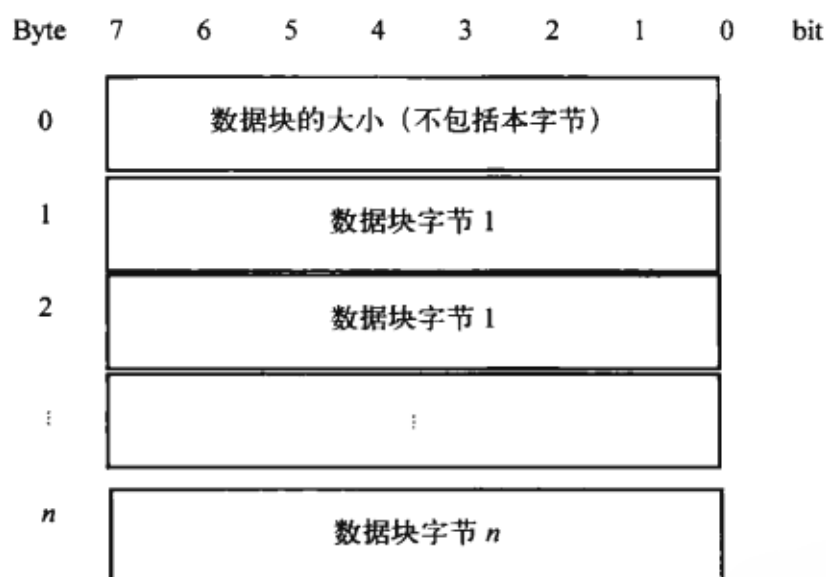


图 3-8 GIF 数据块的结构

一个 GIF 文件的结构可分为文件头 (File Header)、GIF 数据流 (GIF Data Stream) 和文件终结符 (Tailer) 3 个部分。

- 文件头包含 GIF 署名 (Signature) 和版本号 (Version)。
- GIF 数据流由控制标识符、图像块 (Image Block) 和其他的一些扩展块组成。
- 文件终结符只有一个值为  $0x3B$  的字符 “;” 表示文件结束。

图 3-9 显示了一个 GIF 文件的组成结构。



图 3-9 GIF 文件结构

各个部分的具体分析如下。

1. 文件头

GIF 文件头包括 GIF 署名 (Signature) 和版本号 (Version) 两部分。GIF 署名用来确认一个文件是否为 GIF 格式的文件，这一部分由 3 个字符组成：GIF。文件版本号也由 3 个字节组成，可以为 87a 或 89a。GIF 文件版本号 87a 表示 GIF 格式符合 1987 年 5 月发布的 GIF 标准；89a 表示 GIF 格式符合 1989 年 7 月发布的 GIF 标准。

2. GIF 数据流

(1) 逻辑屏幕标识符 (Logical Screen Descriptor)

这一部分由 7 个字节组成，定义了 GIF 图像的大小 (Logical Screen Width and Height)、颜色深度 (Color Bits)、背景色 (Background Color Index)，以及有无全局颜色列表 (Global Color Table) 和颜色列表的索引数 (Index Count)，如图 3-10 所示。

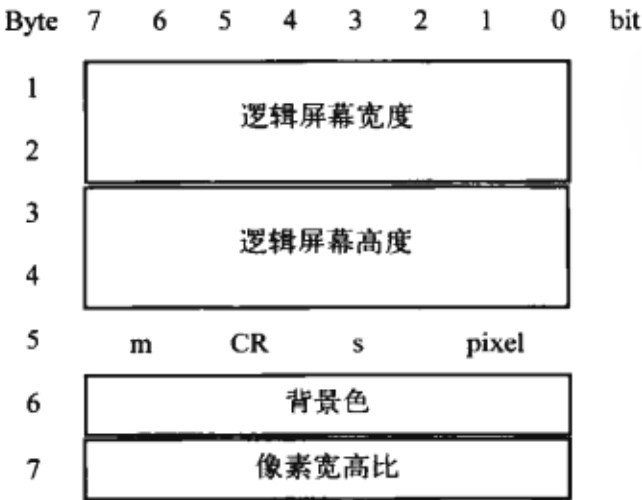


图 3-10 GIF 文件的逻辑屏幕标识符

其中,  $m$  为全局颜色列表标志 (Global Color Table Flag), 当值为 1 时, 表示有全局颜色列表, pixel 值有意义; CR 为颜色深度 (Color Resolution), CR+1 确定图像的颜色深度;  $s$  为分类标志 (Sort Flag), 值为 1 表示全局颜色列表分类排列; pixel 为全局颜色列表大小, pixel+1 确定颜色列表的索引数 ( $2$  的 pixel+1 次方)。

## (2) 全局颜色列表 (Global Color Table)

全局颜色列表必须紧跟在逻辑屏幕标识符后面, 每个颜色列表索引条目由 3 个字节组成, 按 R、G、B 的顺序排列。

## (3) 图像标识符 (Image Descriptor)

一个 GIF 文件内可以包含多幅图像, 故一幅图像结束之后紧接着是下一幅图像的标识符, 图像标识符以 0x2C (“,”) 字符开始, 定义紧接着它的图像的性质, 包括图像相对于逻辑屏幕边界的偏移量、图像大小, 以及有无局部颜色列表和颜色列表大小, 由 10 个字节组成。

其中, 第 1 个字节为二进制 00101100, 即图像标识符的开始标志 0x2C; 第 2、3 个字节为  $X$  方向的偏移量; 第 4、5 字节为  $Y$  方向的偏移量; 第 6、7 字节为图像宽度; 第 8、9 字节为图像高度; 第 10 字节从高位到低位又分为  $m$ 、 $i$ 、 $s$ 、 $r$  和 pixel 五个部分, 其中,  $m$  为局部颜色列表标志 (Local Color Table Flag), 值为 1 时标识紧接在图像标识符之后, 有一个局部颜色列表, 供紧跟在它之后的一幅图像使用, 值为 0 时使用全局颜色列表, 忽略 pixel 值;  $i$  为交织标志 (Interlace Flag), 值为 1 时图像数据使用交织方式排列, 否则使用顺序排列;  $s$  为分类标志 (Sort Flag), 值为 1 表示紧跟着的局部颜色列表分类排列;  $r$  为保留位, 占 2 位, 必须初始化为 0; pixel 为局部颜色列表大小 (Size of Local Color Table), 占 3 位, pixel+1 就是颜色列表的位数。

## (4) 局部颜色列表 (Local Color Table)

如果上面的局部颜色列表标志值为 1, 则需要在这里 (紧跟在图像标识符之后) 定义一个局部颜色列表以供紧接着它的图像使用。使用前应先保存原来的颜色列表, 使用结束之后恢复保存的全局颜色列表。如果一个 GIF 文件既没有提供全局颜色列表, 也没有提供局部颜色列表, 则可以自己创建一个颜色列表, 或使用系统的颜色列表。局部颜色列表也按 RGBRGB... 的方式排列。

## (5) 基于颜色列表的图像数据 (Table-Based Image Data)

图像数据由两部分组成, 即 LZW 最小编码长度 (LZW Minimum Code Size) 和图像数据 (Image Data)。

- LZW 最小编码长度: 表示 GIF 文件使用的 LZW 编码的初始码表大小的位数。
- 图像数据: 由一个或几个数据块 (Data Sub-blocks) 组成。

GIF 图像数据在压缩前有连续的和交织的 (由图像标识符的交织标志控制) 两种排列格式。连续方式按从左到右、从上到下的顺序排列图像的光栅数据。交织图像按下面方法处理光栅数据。

创建 4 个通道 (Pass) 以保存数据, 每个通道提取不同行的数据, 方法如下:

- 第 1 通道 (Pass1) 提取从第 0 行开始每隔 8 行的数据。
- 第 2 通道 (Pass2) 提取从第 4 行开始每隔 8 行的数据。
- 第 3 通道 (Pass3) 提取从第 2 行开始每隔 4 行的数据。
- 第 4 通道 (Pass4) 提取从第 1 行开始每隔 2 行的数据。

GIF 文件的交织图像数据的提取过程如图 3-11 所示。

行	通道 1	通道 2	通道 3	通道 4
0	1			
1				4
2			3	
3				4
4		2		
5				4
6			3	
7				4
8	1			
9				4
10			3	
11				4
12		2		
13				4
14			3	
15				4
16	1			
17				4
18			3	
19				4
20		2		

图 3-11 GIF 文件的交织图像数据的提取

### 3. 文件终结符

GIF 文件的结尾均为一个值为 0x3B 的字节，叫做文件终结符 (Tailer)。

在 GIF 图像的文件结构中，除了上面讨论的几部分以外，在 89a 版本中，还可以有图形控制扩展 (Graphic Control Extension)、注释扩展 (Comment Extension)、图形文本扩展 (Plan Text Extension)、应用程序扩展 (Application Extension) 等。

## 3.3 系统功能

ACDSee 是目前非常流行的一款看图工具，它提供了良好的操作界面，简单人性化的操作方式，优质的快速图形解码方式，支持丰富的图形格式，具有强大的图形文件管理功能。本章讲解的图片浏览器模拟实现 ACDSee 的部分功能。



### 3.3.1 功能描述

图片浏览器主要实现以下功能:

- 1) 打开 BMP、PCX、TGA、JPEG 及 GIF 图像。
- 2) 任意转换以上几种图片的格式。
- 3) 对图片进行逐渐放大、缩小操作,可以直接调整图像显示大小到适合屏幕或恢复为原始状态,可以直接将显示大小调整为原图像的 50%、75%、150%、200%。
- 4) 对打开的图像进行顺时针 90°、逆时针 90°、180° 旋转。
- 5) 直接调用 Windows 画图程序打开图像并进行编辑或以系统默认关联的程序打开图像并编辑。
- 6) 单击“上一张”、“下一张”按钮查看该图片所在文件夹其他图片的功能,即在打开一幅图像后,可以不再使用“打开”命令而用鼠标单击“上一张”、“下一张”按钮或按键盘上的 **Page Up** 键、**Page Down** 键直接浏览当前图片所在文件夹中的其他图片。
- 7) 进行全屏浏览,并在全屏浏览时提供“幻灯片播放”的功能,自动显示当前文件夹下的所有图像。同时,在全屏浏览时,在屏幕右上角显示一个浮动工具条,提供“停止幻灯片播放”、“上一张”、“下一张”、“逐渐放大”、“逐渐缩小”、“适合屏幕大小”、“原始大小”及“退出全屏浏览”的功能。
- 8) 按 **F11** 键可以进行全屏浏览、非全屏浏览的切换,在进行全屏浏览时按 **ESC** 键也可以退出全屏状态。
- 9) 在载入图片时,提供“从上往下”、“从下往上”、“从左往右”、“从右往左”、“左上进入”、“左下进入”、“右上进入”、“右下进入”、“马赛克”、“百叶窗”等显示效果,并且可以由用户选择是否使用及使用哪个效果,用户也可以选择让系统随机选择效果。
- 10) 在查看图片时,用户也可以让图片随时显示“水平百叶窗”、“垂直百叶窗”、“马赛克”、“向上扫描”、“向下扫描”等效果。
- 11) 在查看图片时,可以选择从当前目录中删除该图片,并将其放入系统回收站中。
- 12) 在窗口的用户区右键单击鼠标,则弹出快捷菜单,显示常用的操作命令。
- 13) 在查看图片时,标题栏显示当前打开的图像的文件名;状态栏从左到右依次显示图像的全路径、当前的显示比例、图像文件的大小(KB)、图像的大小、鼠标当前的坐标。

### 3.3.2 界面效果

图 3-12 展示了图片浏览器的运行效果,图中显示的是对 Lena 图顺时针旋转 90° 后的效果。



图 3-12 图片浏览器运行效果

### 3.4 系统结构与流程

图片浏览器主要由图像编码解码模块、图像显示模块和图像变换模块组成，本节给出其总体结构及主要流程。

#### 3.4.1 总体结构

图片浏览器的总体结构如图 3-13 所示。

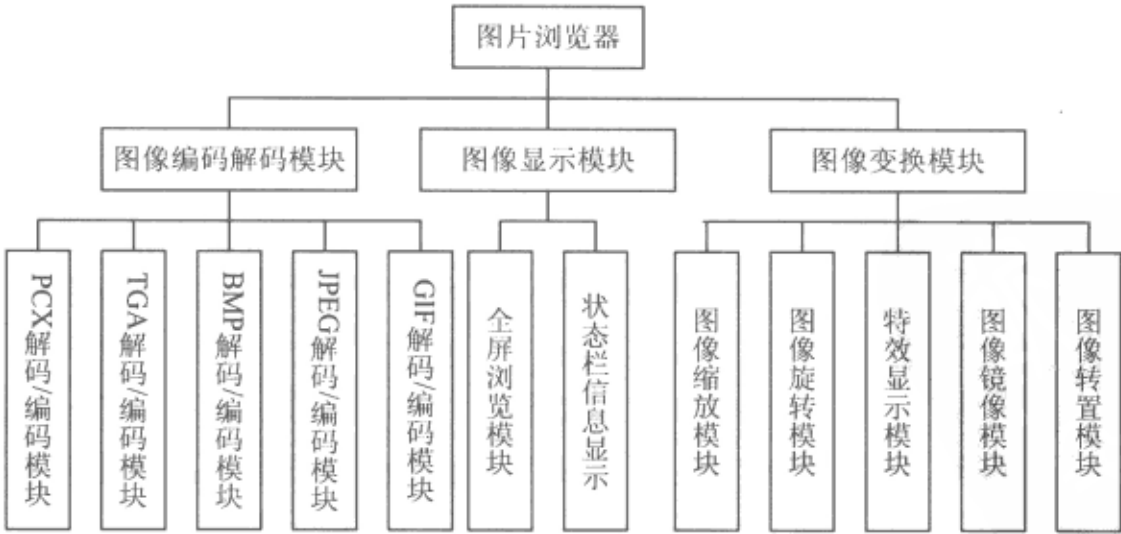


图 3-13 图片浏览器的总体结构

#### 3.4.2 主要流程

图片浏览器的主要流程如图 3-14 所示。

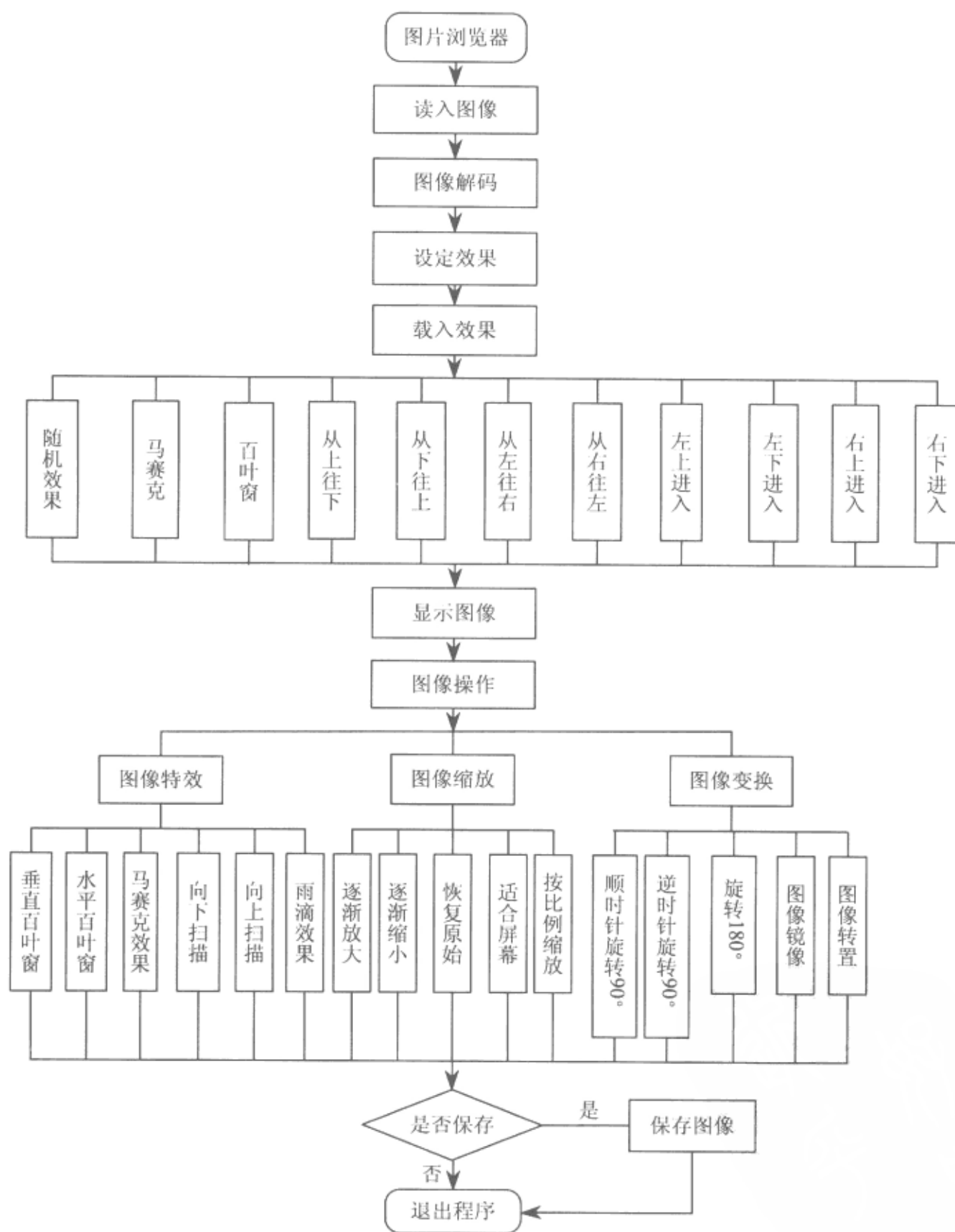


图 3-14 图片浏览器主要流程

## 3.5 编程实现

图片浏览器采用 VC 2008 开发平台编程实现, 自定义了用于存取各种编码格式图像的 LanImage 类。

### 3.5.1 LanImage 类

不同的图像格式, 其解码和编码方式有相当大的区别, 故将不同图像的解码和编码划分为单独的模块; 然而, 不论原图像是何种格式, 在解码之后都会暂时以 RGB 的颜色数据存储在内存中的一块缓冲区中。图像的显示模块及变换模块也就是针对此缓冲区的数据进行操作。因此, 设计一个 LanImage 类, 其中包含一个字节型指针, 指向图像解码后的数据, 该类中还包含图像显示和变换所必需的属性以及对图像进行操作的方法。

由于图像的显示是在视图 (View) 中完成的, 为了便于访问 LanImage 中的方法和属性, 将 LanImage 类作为 View 类 (本程序中为 CPictureBoxView) 的公有成员。

LanImage 类的具体设计代码如下:

```
class LanImage
{
public:
    BYTE * m_pImage;           //指向图像解码后数据缓冲区的指针
    CJPEG* m_pJpeg;            //CJpeg 类对象指针
    CGif* m_pGif;              //CGif 类对象指针
    int m_nWidth;               //图像实际宽度
    int m_nHeight;              //图像实际高度
    int m_nDrawWidth;           //图像显示宽度
    int m_nDrawHeight;          //图像显示高度
    int m_nBitCount;            //图像每像素占用的位数
    int m_nPitch;               //解码后数据一个扫描行所占字节数
    int m_nImage;               //图像数据的字节数
    int m_nPalette;             //调色板中颜色个数
    long nFileSize;             //图像文件大小
    //-----
    BYTE * m_pOriginImage;
    int m_nOriginWidth;
    int m_nOriginHeight;
    int m_nOriginPitch;
    int m_nOriginImage;
    BOOL m_bBufferIsOriginal;
    //-----
    BOOL m_bReadWithLibSupport; //是否使用 JPEG 及 GIF 库中的方法
    RGBQUAD *m_pPal;           //指向调色板区域的指针
};
```

```

LanFormat m_eFmt;           //标志图像数据的存储方式
#ifdef UC_USE_DC             //是否使用 DC, 如果使用则图像解码后
    HBITMAP m_hBitmap;       //的缓冲区使用位图格式
#endif
public:
    LanImage()                //构造函数
    {
        memset(this, 0, sizeof(LanImage));
        m_bReadWithLibSupport=FALSE;
        m_pJpeg=NULL;
        m_pGif=NULL;
        m_pPal=NULL;
    }
    ~LanImage()                //析构函数
    {
        #ifdef UC_USE_DC
            if(m_hBitmap)
                DeleteObject(m_hBitmap);
            //m_hBitmap 中包含了 m_pImage 最初指向的区域
        #else
            if(m_bReadWithLibSupport)
                //如果使用了 Lib Sources, 这时 m_pImage 指
                //向的是原来 CJpeg 类的一部分空间, m_pImage=NULL;
                //而该类已摧毁, 故那部分空间已由其收回
                delete [] m_pImage;
            m_pImage=NULL;
        #endif
        delete m_pJpeg;
        m_pJpeg=NULL;
        delete m_pGif;
        m_pGif=NULL;
        if(m_pPal)
        {
            delete [] m_pPal;
        }
        m_pPal=NULL;
    }

    BOOL LoadImage(LPCTSTR pszFileName, int ndegree, BOOL isconvert);
    //载入图像函数
    BOOL Create(int nWidth, int nHeight, int nBitCount, int ndegree,
        LanFormat eFmt=LanF_Unkown);
    //创建图像解码后的缓冲区
    BOOL ReadBmp(LPBYTE pBuffer, UINT uLength, int ndegree, BOOL isconvert);
    BOOL ReadTga(LPBYTE pBuffer, UINT uLength, int ndegree);
    BOOL ReadPcx(LPBYTE pBuffer, UINT uLength, int ndegree);
    BOOL SaveBmp(LPCTSTR pszFileName) const;
    BOOL SaveTga(LPCTSTR pszFileName) const;

```

```

BOOL SavePcx(LPCTSTR lpstrFileName, CDib* pDib);
//图像显示函数
BOOL Draw(HDC hdc,int nDx,int nDy,int nDw,int nDh,
          int nSx,int nSy,int nSw,int nSh,UINT uFlags=0) const;
BYTE * GetLine(int nLine)      //取得扫描行中一行的数据
{
    return m_pImage+m_nPitch * nLine;
}
}; //类 LanImage

```

### 3.5.2 BMP 解码/编码模块

3.2.1 节已经对 BMP 图像的存储格式进行了分析。BMP 图像的解码由 LanImage 类中的 ReadBmp()函数实现,其中缓冲区的创建调用了 LanImage::Ceate()函数。对于 BMP 图像,去掉 BMP 文件头、信息头之后就是图像的数据信息。可以直接将其读出并放入缓冲区中。

ReadBmp()函数中读取图像数据信息的代码如下:

```

BYTE * pTemp=GetLine(m_nHeight-1);
for(int i=0; i<bih.biHeight; ++i)
{
    memcpy(pTemp,pBuffer,m_nPitch);
    pTemp -=m_nPitch;
    pBuffer +=m_nPitch;
}

```

BMP 的编码,即将图像缓冲区的数据保存为 BMP 格式的功能由 LanImage::SaveBmp()函数完成,其具体实现代码如下:

```

BOOL LanImage::SaveBmp(LPCTSTR pszFileName) const
{
    if(IsLoad()==FALSE)
        return FALSE;
    BITMAPFILEHEADER bfh;
    BITMAPINFOHEADER bih;
    UINT dwMask[3];
    bfh.bfReserved1=0;
    bfh.bfReserved2=0;
    bfh.bfType='MB';
    bfh.bfOffBits=sizeof(bfh)+sizeof(bih)+m_nPalette * sizeof(RGBQUAD);
    memset(&bih,0,sizeof(bih));
    bih.biSize=sizeof(bih);
    bih.biWidth=m_nWidth;
    bih.biHeight=m_nHeight;
    bih.biBitCount=m_nBitCount;
    bih.biPlanes=1;
    bih.biSizeImage=m_nImage;
}

```



```

if(bih.biBitCount==16)
{
    bih.biCompression=BI_BITFIELDS;
    dwMask[0]=0x7C00;
    dwMask[1]=0x03E0;
    dwMask[2]=0x001F;
    bfh.bfOffBits +=sizeof(dwMask);
}
else
    bih.biCompression=BI_RGB;
if(m_nBitCount <=8)
    bih.biClrUsed=m_nPalette;
bfh.bfSize=bfh.bfOffBits+m_nImage;
FILE * fp=fopen(pszFileName,"wb");
//写文件头
fwrite(&bfh,1,sizeof(bfh),fp);
//写信息头
fwrite(&bih,1,sizeof(BITMAPINFOHEADER),fp);
//写调色板
if(m_nPalette)
    fwrite(m_pPal,1,sizeof(RGBQUAD)* m_nPalette,fp);
//写掩码
else if(m_nBitCount==16)
    fwrite(dwMask,1,sizeof(dwMask),fp);
//图像是颠倒的
const BYTE * pImage=GetLine(0);
for(int i=0; i<m_nHeight; ++i)
{
    fwrite(pImage,1,m_nPitch,fp);
    pImage +=m_nPitch;
}
fclose(fp);
return TRUE;
}

```

### 3.5.3 PCX 解码/编码模块

3.2.2 节已分析过 PCX 图像的存储格式，在图片浏览器中，PCX 文件头结构的实现方式是使用一个结构体存储文件头中的各种属性信息，具体如下：

```

typedef struct tagPCXHEAD
{
    BYTE manufacturer;        //0x0A
    BYTE version;             //0,2,3,4,5
    BYTE encoding;            //1 RLE
    BYTE bit_per_pixel;       //1,4,8
    WORD xmin;

```

```

WORD ymin;
WORD xmax;
WORD ymax;
WORD Xresolution;
WORD Yresolution;
BYTE palette[48];           //1 或 4Bits 图像调色板
BYTE reserved;             //0
BYTE color_planes;         //色彩平面数目 (1、3、4)
WORD byte_per_line;        //每行图像的字节宽度
WORD palette_type;         //1
WORD screen_width;
WORD screen_height;
BYTE filter[54];           //0
}PCXHEAD, *PPCXHEAD;

```

PCX 图像的调色板使用如下结构体实现:

```

typedef struct tagPCXPALETTE
{
    BYTE rgbRed;
    BYTE rgbGreen;
    BYTE rgbBlue;
} PCXPALETTE;

```

### 1. PCX 图像的解码

PCX 使用 RLE 压缩编码, 其解码是由 `LanImage::ReadPcx()` 实现的, 当图片每像素位数小于等于 8 时, 调用 `RLE_PCX_Decode8()` 进行解码; 而当图片每像素位数大于 8 时, 调用 `RLE_PCX_Decode24()` 进行解码, 其实现如下:

```

static BYTE * RLE_PCX_Decode8(BYTE * InBuffer, int nInSize, BYTE * OutBuffer)
{
    register BYTE Data, Num;
    while(nInSize > 0)
    {
        Data=*InBuffer++;
        if((Data&0xC0)==0xC0)
        {
            Num=Data&0x3F;
            memset(OutBuffer, *InBuffer++, Num);
            OutBuffer +=Num;
            nInSize -=Num;
        }
        else
        {
            *OutBuffer++=Data;
            nInSize--;
        }
    }
}

```

```

    }
    return InBuffer;
}

static int RLE_PCX_Decode24(BYTE * InBuffer,int nInSize,BYTE * OutBuffer)
{
    register BYTE Data,Num;
    register int nWrite=0;
    while(nInSize-- > 0)
    {
        Data=*InBuffer++;
        if((Data&0xC0)==0xC0)
        {
            Num=Data&0x3F;
            memset(OutBuffer,*InBuffer++,Num);
            OutBuffer +=Num;
            nWrite +=Num;
            nInSize--;
        }
        else
        {
            *OutBuffer++=Data;
            nWrite++;
        }
    }
    return nWrite;
}

```

## 2. PCX 图像的编码

PCX 的编码是由 LanImage::SavePcx()函数完成的,其核心是调用 WritePCXLine()函数,它的功能是将 PCX 的一个扫描行数据写入文件中,其代码如下:

```

BOOL WritePCXLine(int bytes, BYTE *p, FILE *fp)
{
    unsigned int i=0, j=0, t=0;
    do
    {
        i=0;
        while ((p[t+i]==p[t+i+1])&&((t+i)<(WORD)bytes)&&(i<63))
            ++i;
        if (i > 0) //有颜色值相同的相邻像素
        {
            fputc(i|0xc0, fp); //将颜色值相同的相邻像素数写入文件
            fputc(p[t], fp); //将像素值写入文件
            t +=i; //已处理的字节数
            j +=2; //已写入文件的字节数
        }
    }
}

```

```

        else
        {
            if (((p[t])&0xc0)==0xc0)
            {
                fputc(0xc1, fp);    //将 0xc1 标记写入文件
                ++j;
            }
            fputc(p[t++], fp);        //将像素值写入文件
            ++j;
        }
    }
    while (t < (WORD)bytes);
    return (ferror(fp)?FALSE:TRUE);
}

```

### 3.5.4 TGA 解码/编码模块

由于 TGA 图像分为压缩的和非压缩的两类,对于非压缩的 TGA 图像,其读写方法较为简单,与 BMP 类似,将其文件头部分分离后,可直接读取图像数据, TGA 的文件头和文件尾的定义如下:

```

typedef struct tagTGAHEAD
{
    BYTE byID_Length;
    //本结构体大小
    BYTE byPalType;
    //调色板类型: 0 无调色板, 1 有调色板
    BYTE byImageType;
    //01 UC+PAL, 02 UC+NP 03 UC+BW, 09 RLE+PAL, 0A RLE+NP 0B RLE+BW
    WORD wPalFirstNdx;
    //调色板起始索引
    WORD wPalLength;
    //调色板长度
    BYTE byPalBits;
    //调色板中每一颜色占用的位数
    WORD wLeft;//X
    WORD wBottom;//X
    WORD wWidth;
    WORD wHeight;
    BYTE byColorBits;
    BYTE desc;
}TGAHEAD, *PTGAHEAD;
typedef struct tagTGAFooter
{
    Long ext_area;
    Long dev_area;
}

```

```
Char signature[18]; //存放字符串 "TRUEVISION-XFILE"
}TGAFooter;
```

对于压缩的 TGA 图像，由于使用了 RLE 编码，其解码和编码如下所述。

### 1. TGA 图像的解码

TGA 的解码是由 LanImage::ReadTga()函数实现的，其关键是调用解码一行 TGA 图像数据的函数 RLE\_TGA\_DecodeLine()，其代码如下：

```
static BYTE * RLE_TGA_DecodeLine(BYTE * InBuffer,int iColorBit,
int iNumPixel,BYTE * OutBuffer)
{
    register BYTE Data;
    register int Num;
    iColorBit=(iColorBit+7)/8;
    while(iNumPixel > 0)
    {
        Data=*InBuffer++;
        if((Data&0x80)==0x80)
        {
            Num=(Data&0x7F)+1;
            iNumPixel -=Num;
            for(int i=0; i<Num; ++i,OutBuffer +=iColorBit)
                memcpy(OutBuffer,InBuffer,iColorBit);
            InBuffer +=iColorBit;
        }
        else
        {
            Num=++Data;
            iNumPixel -=Num;
            Num *=iColorBit;
            memcpy(OutBuffer, InBuffer, Num);
            OutBuffer +=Num;
            InBuffer +=Num;
        }
    }
    return InBuffer;
}
```

### 2. TGA 图像的编码

TGA 图像的编码是由 LanImage::SaveTga()实现的，其中完成图像数据 RLE 编码的函数为 RLE\_TGA\_EncodeLine()，其代码如下：

```
static BYTE * RLE_TGA_EncodeLine(const BYTE * InBuffer, int iColorBit,
int iNumPixel, BYTE * OutBuffer)
{
    DWORD Data, Next ;
```

## 第3章

```

const BYTE * pBak ;
register DWORD Count ;
iColorBit=(iColorBit+7)/8; //转换为字节数
while (iNumPixel > 0)
{
    pBak=InBuffer ; //记数指针
    memcpy (&Data, InBuffer, iColorBit); //第一个像素
    InBuffer +=iColorBit ;
    iNumPixel-- ;
    Count=1 ;
    while ((Count < 0x7F)&& (iNumPixel > 0)) //统计重复像素
    {
        memcpy (&Next, InBuffer, iColorBit); //下一个像素
        if (Next !=Data)
            break ;
        InBuffer +=iColorBit ;
        iNumPixel-- ;
        Count++ ;
    }
    if (Count==1) //无重复像素
    {
        while ((Count < 0x7F)&& (iNumPixel > 0)) //统计不重复像素
        {
            Count++ ; Data=Next;
            InBuffer +=iColorBit ;
            iNumPixel-- ;
            memcpy (&Next, InBuffer, iColorBit); //下一个像素
            if (Data==Next)
                break ;
        }
        //直接复制不重复像素
        *OutBuffer++=(BYTE) (Count-1);
        Count=InBuffer-pBak ; //Count->临时变量
        memcpy (OutBuffer, pBak, Count);
        OutBuffer +=Count ;
    }
    else //重复像素
    {
        *OutBuffer++=0x80|(BYTE)--Count ;
        memcpy (OutBuffer, &Data, iColorBit);
        OutBuffer +=iColorBit ;
    }
} //End of while
return OutBuffer ;
}

```

### 3.5.5 图像显示模块

图像的显示功能是在 OnDraw() 函数中实现的。当解码模块完成图像格式的解析后, 由 OnDraw() 函数调用 LanImage 类中的 Draw() 函数来将其显示在屏幕的用户区域中。

设计 OnDraw() 函数时要注意以下 3 点:

- 如果用户选择了使用载入特效, 那么在第一次显示时, 应该调用特效显示模块以动画方式将图像显示出来。
- OnDraw() 函数不只在打开图像时被调用, 它同时也是可以被操作系统调用的。比如, 软件的界面被其他窗口遮挡, 当其他的窗口关闭或移开时, 系统会自动调用 OnDraw() 函数, 重绘被遮住的界面。因此, 如果使用特效时不加处理, 那么会在每次软件界面被遮挡并恢复时以动画方式重新显示一次图像, 这会让人感觉很不舒服。
- 如果进行了全屏和非全屏模式的切换, 由于用户区的大小发生了变化, 那么原来图像显示的位置就不是在用户区的中央, 必须重新计算图像的显示位置。

对于以上 3 个问题, 解决的方法是设置标志变量, 在每次执行 OnDraw() 时进行判断, 进而选择合适的操作。在 CPictureBox 类中加入布尔型变量 m\_IsNormalShow, 来标记是否使用特效, 如果该变量为 False 则说明不使用特效, 那么 OnDraw() 函数将不会调用特效显示模块; 在 CPictureBox 类中应用布尔型变量 m\_bIsDrawFirstTime, 来判断是否是第一次进行显示, 如果不是, 那么即使用户选择了使用特效, 也仍然按正常方式显示图像。而对于切换模式后重新计算图像的位置, 可以由以下代码实现:

```
CMainFrame *pFrame=(CMainFrame*)AfxGetApp()->m_pMainWnd;
if(pFrame->m_bScrModeChged) //进行了显示模式切换, 重新计算图片显示位置
{
    CRect rect;
    GetClientRect(&rect); //获取用户区矩形
    m_nWindowWidth=rect.right-rect.left; //计算宽度
    m_nWindowHeight=rect.bottom-rect.top; //计算高度
    //只有当显示的宽、高小于客户区宽、高时才居中
    //否则直接从左上角开始显示, 如果不这样滚动条位置会不正确
    //当显示大小超出客户区时居中也没有意义
    if(m_LImage.m_nDrawWidth<m_nWindowWidth)
        m_showX=m_nWindowWidth/2-m_LImage.m_nDrawWidth/2;
    else
        m_showX=0;
    if(m_LImage.m_nDrawHeight<m_nWindowHeight)
        m_showY=m_nWindowHeight/2-m_LImage.m_nDrawHeight/2;
    else
        m_showY=0;
    pFrame->m_bScrModeChged=FALSE;
}
```



LanImage 类中的 Draw()函数定义如下:

```
BOOL LanImage::Draw(HDC hdc,int nDx,int nDy,int nDw,int nDh,
                    int nSx,int nSy,int nSw,int nSh,UINT uFlags) const
```

此函数调用 MFC 函数 DrawDibDraw()来实现图像的显示,函数各参数的意义如下:

- hdc: 设备 DC 句柄。
- nDx: 图像显示的目标区域左上角 X 坐标。
- nDy: 图像显示的目标区域左上角 Y 坐标。
- nDw: 目标区域宽度。
- nDh: 目标区域高度。
- nSx: 显示时截取源图像区域的左上角 X 坐标。
- nSy: 显示时截取源图像区域的左上角 Y 坐标。
- nSw: 源区域宽度。
- nSh: 源区域高度。
- uFlags: 供 DrawDibDraw()函数使用的标志变量,可忽略。

### 3.5.6 全屏浏览模块

全屏浏览模块是以桌面即 DesktopWindow 的布局来调整软件窗口的布局,从而实现全屏显示的。而全屏时浮动工具栏的自动隐藏,是利用计时器实现的。实现全屏浏览模块的主要代码如下:

```
void CMainFrame::OnFullScreen()
{
    RECT rectDesktop;
    WINDOWPLACEMENT NewWndPlacement; //保存全屏状态的窗口布局
    if (!m_bFullScreen)
    {
        //不需要原状态栏、工具栏
        m_wndStatusBar.ShowWindow(SW_HIDE);
        m_wndToolBar.ShowWindow(SW_HIDE);
        //还需要恢复原状,保存原布局到 m_OldWndPlacement
        GetWindowPlacement(&m_OldWndPlacement);
        m_OldWndPlacement.length=sizeof(m_OldWndPlacement);
        //通过调节 RECT 来确定新窗口的大小
        ::GetWindowRect(::GetDesktopWindow(), &rectDesktop);
        ::AdjustWindowRectEx(&rectDesktop, GetStyle(), TRUE, GetExStyle());
        //把参数用于 OnGetMinMaxInfo()
        m_FullScreenRect=rectDesktop;
        NewWndPlacement=m_OldWndPlacement;
        NewWndPlacement.showCmd=SW_SHOWNORMAL;
        NewWndPlacement.rcNormalPosition=rectDesktop;
```

```

//创建新的工具栏
m_pwndFullScrnBar=new CToolBar;
if(!m_pwndFullScrnBar->CreateEx(this,TBSTYLE_FLAT,WS_CHILD|
    WS_VISIBLE|CBRS_SIZE_DYNAMIC|CBRS_FLOATING |
    CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY)
    || !m_pwndFullScrnBar->LoadToolBar(IDR_FULLSCREEN))
{
    TRACE0("Failed to create toolbar!\n");
    return;    //创建失败
}
//不允许工具栏停靠
m_pwndFullScrnBar->EnableDocking(0);
m_pwndFullScrnBar->SetWindowPos(&CWnd::wndTop,0,100,100,0,
    SWP_NOSIZE|SWP_NOZORDER|SWP_NOACTIVATE|SWP_SHOWWINDOW);
m_pwndFullScrnBar->SetWindowText(_T("全屏/幻灯片模式"));
//得到工具栏大小, 计算其停靠位置
FloatControlBar(m_pwndFullScrnBar,CPoint(100,100));
//如果不先调用一次此函数则取不到其 RECT
RECT FullScrnBarRect;
m_pwndFullScrnBar->GetWindowRect(&FullScrnBarRect);
int BarWidth=FullScrnBarRect.right-FullScrnBarRect.left;
int BarPos_x=rectDesktop.right-rectDesktop.left-BarWidth-15;
FloatControlBar(m_pwndFullScrnBar,CPoint(BarPos_x,3));
//此位置将覆盖 SetWindowPos() 中的设置
m_bFullScreen=TRUE;    //设置全屏显示标志为 TRUE
m_bScrModeChged=TRUE;
//便于在 CPictureView::OnDraw() 中重新计算图片显示位置
//进入全屏时设置计时器及标志以便自动隐藏全屏工具栏
m_bIsFullBarShowed=TRUE;
SetTimer(IDT_TIMER_HIDEFULLSCRBAR,5000,NULL);
}
else    //原来是全屏, 这时再单击“全屏”按钮则应返回正常状态
{
    m_pwndFullScrnBar->DestroyWindow();
    delete m_pwndFullScrnBar;
    m_pwndFullScrnBar=NULL;
    m_bFullScreen=FALSE;
    m_bScrModeChged=TRUE;
    //便于在 CPictureView::OnDraw() 中重新计算图片显示位置
    //如果正在进行幻灯片播放, 则停止
    if(m_bIsPlay)
    {
        m_bIsPlay=FALSE;
        //KillTimer(IDT_TIMER_PLAY);
    }
    //如果全屏工具栏自动隐藏计时器尚未停止, 则退出全屏时停止
    if(m_bIsFullBarShowed && m_bTiming)    //这种情况下计时器在工作

```

```

    {
        m_bIsFullBarShowed=FALSE;
        KillTimer(IDT_TIMER_HIDEFULLSCRBAR);
        m_bTiming=FALSE;
    }
    else
        m_bIsFullBarShowed=FALSE;    //只把标志设为未显示, 不用
                                      //KillTimer, 因Timer早已Kill

    //恢复窗体
    m_wndStatusBar.ShowWindow(SW_SHOWNORMAL);
    m_wndToolBar.ShowWindow(SW_SHOWNORMAL);
    NewWndPlacement=m_OldWndPlacement;
}
SetWindowPlacement(&NewWndPlacement);
}
void CMainFrame::OnGetMinMaxInfo(MINMAXINFO FAR* lpMMI)
{
    //TODO: Add your message handler code here and/or call default
    if (m_bFullScreen)
    {
        lpMMI->ptMaxSize.y=m_FullScreenRect.Height();
        lpMMI->ptMaxTrackSize.y=lpMMI->ptMaxSize.y;
        lpMMI->ptMaxSize.x=m_FullScreenRect.Width();
        lpMMI->ptMaxTrackSize.x=lpMMI->ptMaxSize.x;
    }
    CFrameWnd::OnGetMinMaxInfo(lpMMI);
}

```

### 3.5.7 图像缩放模块

图像的缩放是由 `LanImage` 的 `Draw()` 函数实现的, 由前面的内容可知, `Draw()` 函数提供的参数主要有两部分, 一部分是目标区域的位置和大小; 另一部分是源图像区域的位置和大小。因此, 只要改变目标区域的大小, 即 `nDw` 和 `nDh`, 即可实现图像的缩放。以图像的逐渐缩小为例, 其实现方法如下:

```

void CPictViewView::OnZoomOut()
{
    //TODO: Add your command handler code here
    m_Llower_rate -= 0.1f;    //显示比例逐渐减小, 每次减小 0.1 倍
    if(m_Llower_rate>0)
    {
        m_LImage.m_nDrawWidth=int(m_LImage.m_nWidth*m_Llower_rate);
        m_LImage.m_nDrawHeight=int(m_LImage.m_nHeight*m_Llower_rate);
        if(m_LImage.m_nDrawWidth<m_nWindowWidth)
            m_showX=m_nWindowWidth/2-m_LImage.m_nDrawWidth/2;
        else
    }
}

```

```

        m_showX=0;
        if(m_LImage.m_nDrawHeight<m_nWindowHeight)
            m_showY=m_nWindowHeight/2-m_LImage.m_nDrawHeight/2;
        else
            m_showY=0;
        //设置状态栏“显示比例”
        CMainFrame *pFrame=(CMainFrame *)AfxGetApp()->m_pMainWnd;
        CStatusBar *pStatus=&pFrame->m_wndStatusBar;
        CString sDisplayRate;
        sDisplayRate.Format("显示比例: %.3f",m_Llower_rate);
        pStatus->SetPaneText(1,sDisplayRate);
        m_bIsDrawFirstTime=TRUE;
        Invalidate();                //通知 Windows, 需要重绘
    }
}

```

上述函数中后面部分即为实现状态栏信息显示的改变。本函数中由于改变了图像的显示大小,故“显示比例”一项需要更新,而 m\_Llower\_rate 即为图像当前的显示比例,此变量在 CPictureBoxView 类中定义。

### 3.5.8 图像旋转模块

图像旋转的实现相对较为复杂,在图片浏览器设计中算是个难点。旋转有两种思路,一种是像缩放一样,借助于 LanImage::Draw()函数,按照旋转要求改变目标区域的水平坐标和垂直坐标,使其与源图像的坐标不同。但是要使整幅图像都翻转过来,必须对图像的每个像素调用一次 Draw()函数。这样一来,如果是一幅 640×480 像素大小的图像,旋转一次就要调用 307200 次 Draw()函数,Draw()函数又是调用 DrawDibDraw()实现的,而此函数因为涉及非常复杂的函数调用关系,并不适合进行如此大量次数的调用,如果采用这种方法实现旋转,其执行速度是让人无法忍受的。

因此,要实现旋转,必须在内存中对图像数据的缓冲区先做好变换,然后一次性调用 Draw()函数将其绘制在屏幕上。但是,在内存中进行像素数据的变换,关键之处在于:对每像素位数不同的图像,必须使用不同的操作。例如,如果是 8 位的图片,那么每个像素正好是一个字节,变换时以字节为单位即可;如果是 32 位的图片,则是 4 字节一个像素,那么变换时应以 4 字节为单位;而如果是单色位图,那么 8 个像素才构成一个字节,此时就必须进行位操作。

下面以 8 位和 32 位的图像来说明旋转操作的算法,而需要进行位操作的图像的旋转方法将在 3.6 节进行详细说明,这里不再赘述。

```

void LanImage::Rotate90Clockwise()
{
    BYTE* pNewImage=NULL;

    int iSrcWidth=m_nWidth;

```

```

int iSrcHeight=m_nHeight;
int iSrcRowBytes=m_nPitch;
//计算翻转后一行占用的字节数及数据区的大小
int iNewPitch=(iSrcHeight * m_nBitCount+31)/32 * 4;    //4 字节对齐
int iNewImgDataSize=iNewPitch * iSrcWidth;
//为翻转后的数据分配空间
pNewImage=new BYTE[iNewImgDataSize];
switch(m_nBitCount)
{
    case 8: //8 位图像的翻转
    {
        int OutPointer=0;
        for(int y=0;y < iSrcHeight;y++)
        {
            BYTE *pIn=GetLine(y);
            for(int x=0;x < iSrcWidth;x++)
            {
                //下面这两行看起来是做逆时针旋转,但由于 DIB 是由下到上存储图像的,故实际上是顺时针
                int OutColumn=y;
                int OutRow=iSrcWidth-x-1;
                //而这两行注释掉的则正相反,是逆时针的
                //int OutColumn=iSrcHeight-y-1;
                //int OutRow=x;
                OutPointer=OutRow * iNewPitch+OutColumn;
                CopyMemory(&pNewImage[OutPointer],&pIn[x],1);
            }
        }
        break;
    case 32:
    {
        int OutPointer=0;
        for(int y=0;y < iSrcHeight;y++)
        {
            BYTE *pIn=GetLine(y);
            for(int x=0;x < iSrcWidth;x++)
            {
                //32 位时一个像素占 4 字节宽
                int OutColumn=y*4;
                int OutRow=iSrcWidth-x-1;
                OutPointer=OutRow * iNewPitch+OutColumn;
                CopyMemory(&pNewImage[OutPointer],&pIn[x*4],4);
            }
        }
        break;
    }
}

```

```

}

```

### 3.5.9 图像特效显示模块

与图像的旋转一样，图像的特效显示也有两种实现思路。

- 借助于循环和对 `LanImage::Draw()` 函数的参数进行变换，以达到特效显示的效果。
- 先在内存中将特效的一帧变换好，然后一次性显示整幅图像，再进行下一帧的变换、显示，如此循环直到将所有的帧显示完毕。

显而易见，直接对内存数据操作实现特效的（简称为内存特效）效率要比借助于 `Draw()` 函数的变换高。但是，由于特效所需帧数并不太多，所以使用 `Draw()` 函数来实现也是可行的，并且是现在常用的基本特效实现方法之一；内存特效的实现难度较大。这里采用 `Draw()` 函数参数变换的方法来实现，在 3.5.10 和 3.5.11 节使用内存特效的方法来实现。

以马赛克效果为例。基本操作步骤如下：

- 1) 根据图像大小选择合适的马赛克块的大小。
- 2) 计算出此图像水平和垂直方向可以分的块数。
- 3) 假设水平方向可以分为  $i$  块，垂直方向可以分为  $j$  块，那么创建一个  $i \times j$  大小的整型数组 `LanArrow[i][j]`，初始化为 0。
- 4) 利用随机函数任意选出一块，查看对应的 `LanArrow` 数组元素的值，如果为 0，则显示该块，然后将 `LanArrow` 对应元素置为 1；如果对应的 `LanArrow` 中的值已为 1，则继续随机选择，直到找出一个 `LanArrow` 中值为 0 的块。
- 5) 计数，如果循环次数未达到  $i \times j$ ，返回步骤 4 继续执行，否则结束。

其中，由于 `LanArrow` 的大小事先是无法确定的，需要使用动态二维数组。具体的实现代码如下：

```

void CPictViewView::Effect_Mosaic(CDC *pDC)
{
    int m_count=0;
    int blockNum_w=0;           //水平方向分割的块数或列数
    int blockNum_h=0;           //垂直方向分割的块数或行数
    int blockSize=0;
    if(m_LImage.m_nWidth<=160 || m_LImage.m_nHeight<=160)
        blockSize=10;
    else if(m_LImage.m_nWidth<=400 || m_LImage.m_nHeight<=400)
        blockSize=20;
    else
        blockSize=30;
    //每个块为 blockSize×blockSize 的正方形
    blockNum_w=m_LImage.m_nWidth/blockSize;

```



```

blockNum_h=m_LImage.m_nHeight/blockSize;
int **LanArrow=new int *[blockNum_h];
for(int k=0;k<blockNum_h;k++)
{
    LanArrow[k]=new int [blockNum_w];
}
for(int i=0;i<blockNum_h;i++)
    for(int j=0;j<blockNum_w;j++)
    {
        LanArrow[i][j]=0;
    }
srand( (unsigned)time( NULL ));
while( m_count < blockNum_h*blockNum_w )
{
    int x=rand()%blockNum_w; //随机抽取水平方向上第 x 块, 即第 x 列
    int y=rand()%blockNum_h; //随机抽取垂直方向上第 y 块, 即第 y 行
    while(LanArrow[y][x]==1) //该块已画过。
    {
        x=rand()%blockNum_w;
        y=rand()%blockNum_h;
    }
    LanArrow[y][x]=1;
    m_count++;
    m_LImage.Draw(pDC->GetSafeHdc(),
        int(m_Llower_rate*blockSize*x)+m_showX,
        int(m_Llower_rate*blockSize*y)+m_showY,
        int(m_Llower_rate*blockSize),
        int(m_Llower_rate*blockSize),
        blockSize*x,blockSize*y,blockSize,blockSize);
    Delay(1);
}
for (int n=0; n<blockNum_h; n++)
    delete [blockNum_w] LanArrow[n];
delete [blockNum_h] LanArrow;
//当缩放倍数不为 1 时由于上面强制取整会使图片中有时出现空白线,且当图片的宽、高
//不能恰好分为 8×8 的块时,上面循环过后边界处会有部分未被显示,用下面的语句消除空白
m_LImage.Draw(pDC->GetSafeHdc(),m_showX,m_showY,
int(m_Llower_rate*m_LImage.m_nWidth),
int(m_Llower_rate*m_LImage.m_nHeight),0,0,
m_LImage.m_nWidth,m_LImage.m_nHeight);
}

```

该 Draw()函数实现的特效还有百叶窗左上进入、右上进入等,详细代码请读者参考本书附带的光盘,在此不再赘述。



### 3.5.10 图像镜像模块

下面通过编写 VC 的函数 Mirror() 来实现数字图像的水平镜像, 其具体代码如下:

```

BOOL Mirror(LPSTR lpSrcStartBits, long lWidth, long lHeight, long lLineBytes)
{
    long i; // 行循环变量
    long j; // 列循环变量
    LPSTR lpSrcDIBBits; // 指向源像素的指针
    LPSTR lpDstDIBBits; // 指向临时图像对应像素的指针
    HLOCAL hDstDIBBits; // 临时图像句柄
    LPSTR lpBits; // 指向中间像素的指针, 当复制图像时, 提供临时的像素内存空间
    hDstDIBBits = LocalAlloc(LHND, lLineBytes); // 分配临时内存, 保存行图像
    if (hDstDIBBits == NULL)
        return FALSE; // 分配内存失败
    lpDstDIBBits = (char *) LocalLock(hDstDIBBits); // 锁定
    for (i = 0; i < lHeight; i++) // 水平镜像, 针对图像每行进行操作
    {
        for (j = 0; j < lWidth / 2; j++) // 针对每行图像左半部分进行操作
        {
            // 指向倒数第 i 行, 第 j 个像素的指针
            lpSrcDIBBits = (char *) lpSrcStartBits + lLineBytes * i + j;
            // 指向倒数第 i+1 行, 倒数第 j 个像素的指针
            lpBits = (char *) lpSrcStartBits + lLineBytes * (i + 1) - j;
            *lpDstDIBBits = *lpBits; // 保存中间像素
            // 将倒数第 i 行, 第 j 个像素复制到倒数第 i 行, 倒数第 j 个像素
            *lpBits = *lpSrcDIBBits;
            // 将倒数第 i 行, 倒数第 j 个像素复制到倒数第 i 行, 第 j 个像素
            *lpSrcDIBBits = *lpDstDIBBits;
        }
    }
    LocalUnlock(hDstDIBBits); // 释放内存
    LocalFree(hDstDIBBits);
    return TRUE;
}

```

通过编写 VC 的函数 Mirror2() 来实现数字图像的垂直镜像, 其具体代码如下:

```

BOOL Mirror2(LPSTR lpSrcStartBits, long lWidth, long lHeight, long lLineBytes)
{
    long i; // 行循环变量
    LPSTR lpSrcDIBBits; // 指向源像素的指针
    LPSTR lpDstDIBBits; // 指向临时图像对应像素的指针
    HLOCAL hDstDIBBits; // 临时图像句柄
    LPSTR lpBits; // 指向中间像素的指针, 当复制图像时, 提供临时的像素内存空间
    hDstDIBBits = LocalAlloc(LHND, lLineBytes); // 分配临时内存, 保存行图像
    if (hDstDIBBits == NULL)

```

```

        return FALSE;    // 分配内存失败
lpDstDIBBits= (char *)LocalLock(hDstDIBBits);    // 锁定
for(i = 0; i < lHeight / 2; i++)    // 垂直镜像, 针对图像每行进行操作
{
    // 指向倒数第 i 行, 第 j 个像素的指针
    lpSrcDIBBits= (char *)lpSrcStartBits + lLineBytes * i ;
    // 指向倒数第 i+1 行, 倒数第 j 个像素的指针
    lpBits= (char *)lpSrcStartBits + lLineBytes * (lHeight - i + 1);
    memcpy(lpDstDIBBits, lpBits, lLineBytes);
    memcpy(lpBits, lpSrcDIBBits, lLineBytes);
    memcpy(lpSrcDIBBits, lpDstDIBBits, lLineBytes);
}
LocalUnlock(hDstDIBBits);    // 释放内存
LocalFree(hDstDIBBits);
return TRUE;
}

```

### 3.5.11 图像转置模块

下面通过编写 VC 的函数 Transpose() 来实现数字图像的转置, 其具体代码如下:

```

BOOL Transpose(LPSTR lpSrcDib, LPSTR lpSrcStartBits, long lWidth, long lHeight,
               long lLineBytes, long lDstLineBytes)
{
    long i;    // 行循环变量
    long j;    // 列循环变量
    LPSTR lpSrcDIBBits;    // 指向源像素的指针
    LPSTR lpDstDIBBits;    // 指向临时图像对应像素的指针
    LPSTR lpDstStartBits;    // 指向临时图像对应像素的指针
    HLOCAL hDstDIBBits;    // 临时图像句柄
    LPBITMAPINFOHEADER lpbmi;    // 指向 BITMAPINFOHEADER 结构的指针
    lpbmi = (LPBITMAPINFOHEADER)lpSrcDib;
    hDstDIBBits= LocalAlloc(LHND, lWidth * lDstLineBytes);    // 分配临时内存
    if (hDstDIBBits== NULL)    // 判断是否内存分配
        return FALSE;    // 分配内存失败
    lpDstStartBits= (char *)LocalLock(hDstDIBBits);    // 锁定内存
    for(i = 0; i < lHeight; i++)    // 针对图像每行进行操作
    {
        for(j = 0; j < lWidth; j++)    // 针对每行图像每列进行操作
        {
            // 指向源 DIB 第 i 行, 第 j 个像素的指针
            lpSrcDIBBits= (char *)lpSrcStartBits + lLineBytes * (lHeight - 1 - i) + j;
            // 指向转置 DIB 第 j 行, 第 i 个像素的指针
            lpDstDIBBits= (char *)lpDstStartBits + lDstLineBytes * (lWidth - 1 - j) + i;
            *(lpDstDIBBits)= *(lpSrcDIBBits);    // 复制像素
        }
    }
}

```

```

        memcpy(lpSrcStartBits, lpDstStartBits, lWidth * lDstLineBytes);
        // 复制转置后的图像
        lpbmi->biWidth = lHeight;
        lpbmi->biHeight = lWidth;
        LocalUnlock(hDstDIBBits);    // 释放内存
        LocalFree(hDstDIBBits);
        return TRUE;
    }

```

### 3.6 经验分享

在图片浏览器编程实现的过程中,资源的分配和释放、实现旋转时对内存的位操作,以及浏览功能的实现容易出问题,需要仔细处理。

1) 资源的分配和释放。由于 Visual C++ 中内存资源的分配和释放均由程序员控制,而图片浏览器程序在内存中开辟多处缓冲区,因此对内存的管理尤为重要。

比如,在打开一张新图片时,就要对上一张图片的缓冲区进行释放,否则会造成内存的泄漏。但是由于 JPEG 和 GIF 调用了第三方的读写库,其使用的内存缓冲区格式与 LanImage 类中使用的不同,因此,在进行缓冲区的释放时,首先要判断上一张图片是何种格式,即是否使用了第三方的读写库,如果使用了,则应采取不同的资源释放方式。

图片浏览器程序是通过在 LanImage 类中加入布尔型变量 m\_bReadWithLibSupport 来判断是否使用了 JPEG 和 GIF 读写库的。例如,在 OnFileOpen() 函数中,资源的释放代码如下:

```

if (m_LImage.m_bReadWithLibSupport)
{
    //在 LanImage::LoadImage() 中 m_pJpeg=new CJpeg() 分配的空间
    if (m_LImage.m_pJpeg !=NULL)    //上次打开的是 JPEG 格式
    {
        CJpeg * pTempJpeg=m_LImage.m_pJpeg;
        m_LImage.m_pJpeg=NULL;
        if(!m_LImage.m_bBufferIsOriginal)    //进行过变换
            delete [] m_LImage.m_pImage;
        m_LImage.m_pImage=NULL;
        m_LImage.m_pOriginImage=NULL;
        delete pTempJpeg;
    }
    if(m_LImage.m_pGif !=NULL)    //上次打开的是 GIF 格式
    {
        CGif * pTempGif=m_LImage.m_pGif;
        m_LImage.m_pGif=NULL;
        if(!m_LImage.m_bBufferIsOriginal)    //进行过变换
            delete [] m_LImage.m_pImage;
    }
}

```

```

        m_LImage.m_pImage=NULL;
        m_LImage.m_pOriginImage=NULL;
        delete pTempGif;
    }
}
else
    m_LImage.Release();

```

2) 实现旋转时对内存的位操作。3.1.2 节已介绍过旋转的算法, 而单色和 4 位色图像进行旋转时, 需要进行位操作。以单色位图为例, 一个字节包含 8 个像素, 那么在旋转  $90^\circ$  时, 一个字节中的不同位, 会变换到不同的扫描行, 也就是说要将字节中的位一一取出, 计算其位置, 并重新放到新字节的合适的位置中, 代码如下:

```

void LanImage::Rotate90Clockwise()
{
    BYTE* pNewImage=NULL;

    int iSrcWidth=m_nWidth;
    int iSrcHeight=m_nHeight;
    int iSrcRowBytes=m_nPitch;
    //计算翻转后一行占用的字节数及数据区的大小
    int iNewPitch=(iSrcHeight * m_nBitCount+31)/32 * 4;    //4 字节对齐
    int iNewImgDataSize=iNewPitch * iSrcWidth;
    //为翻转后的数据分配空间
    pNewImage=new BYTE[iNewImgDataSize];
    switch(m_nBitCount)
    {
        case 1:
        {
            //全置为 0, 便于后面的按位或操作
            memset(pNewImage,0,iNewImgDataSize);
            int OutPointer=0;
            for(int y=0;y<iSrcHeight;y++)
            {
                BYTE* pIn=GetLine(y);
                for(int x=0;x<iSrcRowBytes;x++)
                {
                    int BitPointer=0;    //确定是在该字节中的第几位
                    int OutColumn=y/8;    //8 行翻转后才构成一字节
                    BYTE byTempIn=pIn[x];    //取一个字节, 即 8 个像素
                    BYTE byTempOut;
                    for(BitPointer=0;BitPointer<8;BitPointer++)
                    {
                        int OutRow=iSrcWidth-(x*8+BitPointer)-1;
                        //注意: 最高位代表最左边的像素
                        switch(BitPointer)
                        {

```

```

        case 0:
            byTempOut=(byTempIn&0x80);    //1000 0000
            //得出此位在目的字节中应在哪一位
            byTempOut=(byTempOut >> (y%8));
            break;
        case 1:
            byTempOut=(byTempIn&0x40);    //0100 0000
            if(y%8-1 < 0)
                byTempOut=(byTempOut << (1-y%8));
            else
                byTempOut=(byTempOut >> (y%8-1));
            break;
        case 2:
            byTempOut=(byTempIn&0x20);    //0010 0000
            if(y%8-2 < 0)
                byTempOut=(byTempOut << (2-y%8));
            else
                byTempOut=(byTempOut >> (y%8-2));
            break;
        case 3:
            byTempOut=(byTempIn&0x10);    //0001 0000
            if(y%8-3 < 0)
                byTempOut=(byTempOut << (3-y%8));
            else
                byTempOut=(byTempOut >> (y%8-3));
            break;
        case 4:
            byTempOut=(byTempIn&0x08);    //0000 1000
            if(y%8-4 < 0)
                byTempOut=(byTempOut << (4-y%8));
            else
                byTempOut=(byTempOut >> (y%8-4));
            break;
        case 5:
            byTempOut=(byTempIn&0x04);    //0000 0100
            if(y%8-5 < 0)
                byTempOut=(byTempOut << (5-y%8));
            else
                byTempOut=(byTempOut >> (y%8-5));
            break;
        case 6:
            byTempOut=(byTempIn&0x02);    //0000 0010
            if(y%8-6 < 0)
                byTempOut=(byTempOut << (6-y%8));
            else
                byTempOut=(byTempOut >> (y%8-6));
            break;

```

```

        case 7:
            byTempOut=(byTempIn&0x01);    //0000 0001
            byTempOut=(byTempOut << (7-y%8));
            break;
        }
        OutPointer=OutRow * iNewPitch +OutColumn;
        BYTE byDestBuffer=pNewImage[OutPointer];
        byDestBuffer=(byDestBuffer|byTempOut);
        pNewImage[OutPointer]=byDestBuffer;
    } //for(BitPointer)
    } //for(x)
    } //for(y)
}
break;
.....
}

```

3) 浏览功能的实现。在打开一幅图片后,可以使用“上一张”命令、“下一张”命令或按“Page Up”键、“Page Down”键浏览当前目录中可支持的图像。本功能的实现关键在于打开一幅图片时,读取图片所在目录的文件列表,并对其进行分析,如果有可支持的图像,则记录到一个二维数组中。在打开一幅图片并读取当前目录中的图像列表后,如果用户对该目录中的图片进行了删除或向该目录中加入了新的图片,那么在使用“上一张”命令和“下一张”命令时,应该重新读取文件列表。

读取列表功能由 CPictViewDoc::InitPicList()函数实现,其核心代码如下:

```

if(pszExt)
//如果有扩展名,则继续分析扩展名,否则不分析
{
    m_PicList[m_iFileNuif((_tcsicmp(pszExt, ".bmp")==0)
        ||(_tcsicmp(pszExt, ".tga")==0)
        ||(_tcsicmp(pszExt, ".pcx")==0)
        ||(_tcsicmp(pszExt, ".jpg")==0)
        ||(_tcsicmp(pszExt, ".jpeg")==0)
        ||(_tcsicmp(pszExt, ".gif")==0))
    {
        m_PicList[m_iFileNumber]=cFind.GetFileName();
        CString FilePath;
        FilePath.Format("%s%s",drive,dir);
        m_PicList[m_iFileNumber]=FilePath+m_PicList[m_iFileNumber] ;
        m_iFileNumber++;
    }
}

```



## 第 4 章 图像编辑器

丑小鸭化身白天鹅，那是安徒生童话世界里对美的追求和向往，而把灰姑娘 PS 成天使，则是图像编辑软件中对图像进行的美化加工。图像编辑、合成、调色及特效制作等是平面设计、婚纱摄影等领域中常用的图像处理功能。其中，图像编辑是基础，主要是对图像进行一些变换和增强处理。

**本章要点：**

- 灰度变换增强技术
- 直方图增强技术
- 平滑去噪技术
- 图像锐化技术
- 模糊复原技术
- 彩色增强技术
- 滤镜技术
- 图像编辑器功能描述
- 图像编辑器的总体结构和主要流程
- 图像编辑器的编程实现

### 4.1 核心技术原理

一个合格的图像编辑器需要的技术有灰度变换增强技术、直方图增强技术、彩色增强技术、平滑去噪技术、图像锐化技术、模糊复原技术等。

#### 4.1.1 灰度变换增强技术

图像的灰度变换增强是图像增强处理技术中一种简单、直接的基于空间域的图像处理方法。在图像处理中，空间域是指由像素组成的空间，空间域增强方法是指直接作用于图像像素的增强方法。空间域处理可表示为：



$$g(x, y) = T[f(x, y)] \quad (4-1)$$

其中  $f(x, y)$  是增强前的图像,  $g(x, y)$  是增强处理后的图像, 而  $T$  是对  $f$  的一种操作, 其定义在  $(x, y)$  的邻域上。如果  $T$  是定义在每个  $(x, y)$  点上, 则  $T$  称为点操作; 如果  $T$  是定义在  $(x, y)$  的某个邻域上, 则  $T$  称为模板操作。

$T$  操作最简单的形式是邻域为  $1 \times 1$  的尺度 (即单个像素), 在这种情况下,  $g$  的值仅仅依赖  $f$  在  $(x, y)$  点的值,  $T$  操作称为灰度变换函数。灰度变换函数描述了输入灰度值与输出灰度值之间的转换关系。一旦灰度变换函数确定, 则图像中每一点的运算就被完全确定下来。

### 1. 线性灰度增强

图像的线性灰度增强就是将图像中所有点的灰度按照线性灰度变换函数进行变化。在曝光不足或者过度的情况下, 图像的灰度可能局限在一个很小的范围, 这时图像显示出来将会模糊不清。用一个线性单值函数, 对图像内的每一个像素做线性扩展, 将会有效地改善图像的视觉效果。

假设变换前图像  $f(x, y)$  的灰度范围为  $[a, b]$ , 希望变换后的图像  $g(x, y)$  的灰度范围扩展或压缩至  $[c, d]$ , 则灰度线性变换函数表达式为:

$$g(x, y) = [(d - c) / (b - a)](f(x, y) - a) + c \quad (4-2)$$

可以通过调整  $a, b, c, d$  的值控制线段的斜率从而对图像灰度区间进行扩展与压缩。

### 2. 分段线性灰度增强

分段线性灰度增强将需要的图像细节灰度级扩展, 增强对比度, 将不需要的图像细节灰度级压缩。常用的方法是分 3 段做线性灰度变换。

假设输入图像  $f(x, y)$  的灰度为  $0 \sim M$  级, 增强后图像  $g(x, y)$  的灰度为  $0 \sim N$  级, 区间  $[a, b]$ 、 $[c, d]$  分别为原图形与增强图像的某一灰度区间。分段线性变换函数为:

$$g(x, y) = \begin{cases} (c/a)f(x, y) & 0 \leq f(x, y) < a \\ [(d - c)/(b - a)][f(x, y) - a] + c & a \leq f(x, y) \leq b \\ [(N - d)/(M - b)][f(x, y) - b] + d & b < f(x, y) \leq M \end{cases} \quad (4-3)$$

实际上  $a, b, c, d$  可取不同的值进行组合, 从而得到不同的效果。

若  $a = c, b = d$ , 则灰度变换函数  $T$  为一条斜率为 1 的直线, 增强图像将和原图像相同。

若  $a > c, b < d$ , 则原图像中灰度值在区间  $[0, a]$  与  $[b, M]$  中的动态范围减小了, 而原图像在区间  $[a, b]$  间的动态范围增加了, 从而增强了中间范围的对比度。

若  $a < c, b > d$ , 则原图像在区间  $[0, a]$  与  $[b, M]$  中的动态范围增加了, 而原图像在区间  $[a, b]$  间的动态范围减小了。由此可见通过调整  $a, b, c, d$  可以控制分段的斜率, 从而对任意灰度区间进行扩展和压缩。

### 3. 非线性灰度增强

在前面的灰度变换中,输入图像与增强图像之间是一种线性变换关系,当用某些非线性函数对图像灰度进行映射时可以实现图像灰度的非线性变换。

常用的图像灰度非线性变换有对数函数非线性变换和指数函数非线性变换,下面分别介绍一下它们的原理。

#### (1) 对数函数非线性变换

对图像做对数函数非线性变换时,变换函数为:

$$g(x,y) = \frac{\ln[f(x,y)+1]}{b \cdot \ln c} \quad (4-4)$$

通过调整参数  $a$ ,  $b$ ,  $c$  可以调整曲线的位置与形状。利用此变换可以使输入图像的低灰度范围得到扩展,高灰度范围得到压缩,以使图像分布均匀。

#### (2) 指数函数非线性变换

对图像做指数函数非线性变换时,变换函数为:

$$g(x,y) = bc[f(x,y)-a]-1 \quad (4-5)$$

通过调整参数  $a$ ,  $b$ ,  $c$  可以调整曲线的位置与形状。利用此变换可以使输入图像的高灰度范围得到扩展,低灰度范围得到压缩,以使图像分布均匀。

## 4.1.2 直方图增强技术

图像直方图是图像处理中一种十分重要的图像分析工具,它描述了一幅图像的灰度级内容,任何一幅图像的直方图都包含了丰富的信息。图像直方图灰度级的分布形态可以提供图像信息的许多特征,所以可以通过改变直方图的形状来达到增强图像对比度的效果。常用的方法有直方图均衡化与直方图规定化。

### 1. 直方图统计

直方图为图像的处理提供了一个有力的辅助工具。可以通过直方图的显示来判断一幅图像是否合理地利用了全部允许使用的灰度范围,通过直方图了解图像的灰度分布,通过对图像灰度密度的修改,有选择地突出所需要的图像特征以满足需求。

从数学上来说图像直方图是图像各灰度值统计特性与图像灰度值之间的函数,它统计一幅图像中各个灰度级出现的次数或概率。从图形上来说,它是一个二维图,横坐标是图像中各个像素点的灰度级,用  $\gamma$  表示,纵坐标为具有该灰度级的像素个数或出现这个灰度级的概率  $P(\gamma_k)$ 。

$$P(\gamma_k) = n_k / N \quad (4-6)$$

其中  $N$  为一幅图像中像素的总数,  $n_k$  为第  $k$  级灰度的像素数,  $\gamma_k$  表示第  $k$  个灰度级,  $P(\gamma_k)$  表示该灰度级出现的概率。因为  $P(\gamma_k)$  给出了对  $\gamma_k$  出现概率的估计,因此直方图提供了图像的灰

度值分布情况,也就是说给出了图像灰度值的整体描述。

## 2. 直方图均衡化

若一幅图像其像素占有全部可能的灰度级并分布均匀,那么这样的图像有高对比度和多变的灰度色调,从而显示出来的图像就会灰度级丰富并且动态范围很大。直方图均衡的基本思想是对原始图像中的像素灰度做某种映射变换,使变换后的图像灰度的概率密度均匀分布,即变换后图像是一幅灰度级均匀分布的图像,这就意味着图像灰度的动态范围得到增加,对比度得到提高。

假设  $\gamma$ ,  $s$  分别代表原始图像和变换后的图像在点  $(x, y)$  处的灰度值,灰度级总数为  $L$  个,且  $s = T(\gamma)$ ,  $T(\gamma)$  为变换函数。这里图像增强变换函数需要满足以下 2 个条件:

条件 1:  $T(\gamma)$  在  $0 \leq \gamma \leq L-1$  范围内是个单值单增的函数。这样保证图像灰度级在变换后仍保持从黑到白的排列次序。

条件 2: 对  $0 \leq \gamma \leq L-1$  有  $0 \leq T(\gamma) \leq L-1$ 。这样保证变换前后灰度值动态范围的一致性。

由  $s$  到  $\gamma$  的逆变换为:

$$\gamma = T^{-1}(s) \quad (0 \leq s \leq L-1) \quad (4-7)$$

这里  $T^{-1}(s)$  对  $s$  也满足条件 1 与条件 2。

一幅图像的灰度级可以视为区间  $[0, L-1]$  上的随机变量,可以证明变换函数是原图像的累积分布函数,并满足以上两个条件。假设  $N$  为一幅图像中像素的总数,  $n_k$  为第  $k$  级灰度的像素数,  $\gamma_k$  表示第  $k$  个灰度级,则该图像中灰度级为  $\gamma_k$  的像素出现的概率为:

$$P(\gamma_k) = n_k / N \quad (k=0,1,2,\dots,L-1) \quad (4-8)$$

对其进行均匀化处理后的变换函数为:

$$s_k = T(\gamma_k) = \sum_{j=0}^k P(r_j) = \sum_{j=0}^k \frac{n_j}{N} \quad (4-9)$$

利用上式对图像做灰度变换,即可得到直方图均衡化后的图像。

## 3. 直方图规定化

直方图均衡化虽然使图像的对比度得到增强,但是这种增强是一种整体的变化,处理结果得到全局均衡化的直方图。而在实际应用时往往需要将直方图变换为某种特定的形状,以实现输入图像进行有目的地增强。

为了实现对输入图像进行有目的地增强,就需要选择合适的变换函数。一般来说,正确地选择规定化的函数可以获得比直方图均衡化更好的效果。因为规定化变换是为了满足不同的处理需要而提出的,所以对不同的图像,甚至对同一幅图像的规定化函数都是不同的,在调用规定化变换函数时,必须指定规定化变换后的直方图形式。

### 4.1.3 平滑去噪技术

图像在采集、传输、处理过程中往往会存在一定程度的噪声干扰，噪声恶化了图像的质量，使图像模糊，特征被淹没，给图像分析带来困难。因此去除噪声是图像处理的一个重要内容。

图像平滑是一种实用的图像处理技术，能消除图像采集、传输、处理过程中的噪声。图像平滑包括空域法和频域法两大类，在空域中主要利用邻域平均法、加权平均法、选择式掩膜平滑法和中值滤波法来消除图像噪声；在频率域中主要利用各种形式的低通滤波器来消除噪声。本节主要介绍几种常用的空域图像平滑方法。

#### 1. 邻域平均法

噪声点像素的灰度与它们临近像素的灰度有着显著的不同，根据噪声点这一特性，可以使用邻域平均法。邻域平均法采用模板计算的思想，模板操作实现了一种邻域运算，它用模板确定的邻域内的像素值去代替图像中每一个像素点的值。

设  $f(i, j)$  为给定的含有噪声的图像，经过邻域平均法处理后的图像为  $g(i, j)$ ，在数学上表现为： $g(i, j) = \sum f(i, j) / N$ ， $(i, j) \in S$ ， $S$  所取邻域中的各临近像素的坐标， $N$  是邻域中包含的临近像素个数。

在实际应用中，可以根据不同需要选择不同的模板尺寸，如  $3 \times 3$ 、 $5 \times 5$ 、 $7 \times 7$  等。常用的一个  $3 \times 3$  的模板为： $\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ ，模板中心对应的是需要进行处理的像素。

#### 2. 加权平均法

邻域平均处理方法是以前图像模糊为代价来减小噪声的。有时候为了突出原图像中的点  $(i, j)$  本身的重要性，对于同一尺寸的模板，不同位置的系数采用不同的数值，这就需要采用加权平均法。

一般认为离对应模板中心像素近的像素对平滑结果有比较大的影响，所以接近模板中心的系数应较大，而模板边界附近的系数应较小。

常用的加权平均模板有： $\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$ ， $\frac{1}{10} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix}$  等。

#### 3. 选择式掩膜平滑法

邻域平均法和加权平均法在消除噪声的同时，都不可避免地带来平均化的缺憾，致使尖锐变化的边缘或线条变得模糊。考虑到图像中目标物体和背景一般都具有不同的统计特性，即不同的均值和方差，为保留一定的边缘信息，可采用自适应的局部平滑滤波，这样可以得到较好的图像细节。自适应平滑方法以尽量不模糊边缘轮廓为目的，即选择式掩膜平滑法。

选择式掩膜平滑法取  $5 \times 5$  的模板窗口, 如图 4-1 所示。在窗口内以中心像素  $(i, j)$  为基准点, 制作 4 个五边形、4 个六边形、一个边长为 3 的正方形共 9 个形状的画面窗口, 分别计算每个窗口内的平均值及方差。由于含有尖锐边缘的区域, 方差必定比平缓区域大, 因此采用方差最小的屏蔽窗口进行平均化, 这种方法在完成滤波操作的同时, 又不破坏区域边界的细节。这种采用 9 种形状的屏蔽窗口, 分别计算各窗口内的灰度值方差, 并采用方差最小的屏蔽窗口进行平均化的方法, 也称为自适应平滑方法。

根据图 4-1 的各种模板分别计算各模板作用下的均值及方差。均值计算公式为:

$$M_i = \frac{\sum_{k=1}^N f(i, j)}{N} \quad (4-10)$$

方差计算公式为:

$$\sigma_i = \sum_{k=1}^N (f^2(i, j) - M^2) \quad (4-11)$$

式中,  $k=1, 2, 3, \dots, N$ ,  $N$  为各掩膜对应的像素个数。

将计算得到的方差进行排序, 最小方差所对应的掩膜的灰度级均值  $M_i$  作为平滑结果输出。用同样的方法作用于每个像素, 即可完成图像的平滑操作。

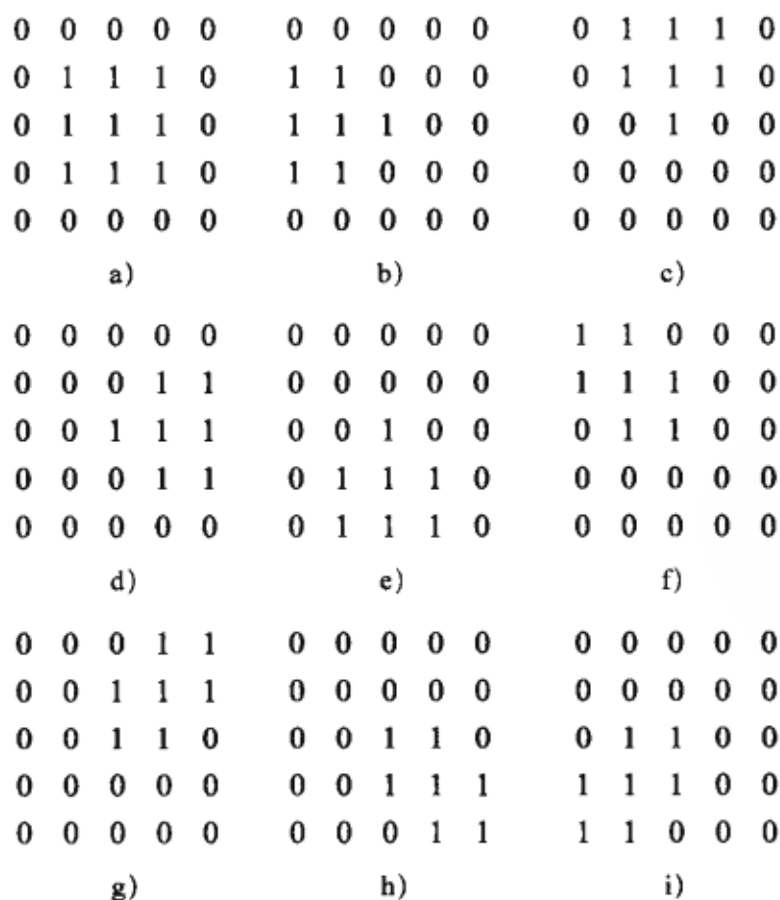


图 4-1 9 种屏蔽窗口的模板

#### 4. 中值滤波法

中值滤波是一种非线性平滑滤波，在一定条件下可以克服线性滤波所带来的图像细节模糊问题，而且对滤除噪声干扰及图像扫描噪声非常有效。

中值滤波通常是用一个有奇数个点的滑动窗口，用窗口中各点灰度值的中值来代替中心点的灰度值。对于奇数个元素，中值是指按大小排序后中间的数值；对偶数个元素，中值是指排序后中间两个元素灰度值的平均值。常用的中值滤波窗口形状有线状、方形、圆形、十字形等。

##### 4.1.4 图像锐化技术

图像锐化就是补偿图像的轮廓，增强图像的边缘及灰度跳变的部分，使图像变得清晰。图像平滑往往使图像中的边界、轮廓模糊，为了减少这类不利效果的影响，就需要利用图像锐化技术使图像的边缘清晰。图像锐化处理的目的是为了使图像的边缘、轮廓线以及图像的细节变得清晰，经过平滑的图像变得模糊的根本原因是因为对图像进行了平均或积分运算，因此可以对其进行逆运算（如微分运算）就可以使图像变得清晰。下面介绍两种常用的锐化方法：梯度锐化和拉普拉斯掩膜锐化。

##### 1. 梯度锐化

对图像  $f(x, y)$ ，在其点  $(x, y)$  处的梯度是一个矢量：

$$G[f(x, y)] = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \quad (4-12)$$

梯度的方向在函数  $f(x, y)$  最大变化率的方向上，梯度的幅度  $G_M[f(x, y)]$  可由下式给出：

$$G_M[f(x, y)] = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \quad (4-13)$$

由上式可知，梯度的数值就是  $f(x, y)$  在其最大变化率方向上的单位距离所增加的量。

对数字图像而言，微分  $\partial f / \partial x$  和  $\partial f / \partial y$  可用差分来近似。差分运算近似后的梯度表达式为：

$$G_M[f(x, y)] = \sqrt{[f(x, y) - f(x+1, y)]^2 + [f(x, y) - f(x, y+1)]^2} \quad (4-14)$$

对图像  $f(x, y)$ ，上式的计算量很大，因此在实际计算中常用绝对值代替平方和平方根运算，梯度模

$$G_M[f(x, y)] = |f(x, y) - f(x+1, y)| + |f(x, y) - f(x, y+1)| \quad (4-15)$$

这种梯度法称为水平垂直差分法，另一种梯度法是交叉地进行差分计算，称为罗伯特梯度法，表示为：

$$G_M[f(x, y)] = \sqrt{[f(x, y) - f(x+1, y+1)]^2 + [f(x+1, y) - f(x, y+1)]^2} \quad (4-16)$$

同样，用绝对值代替平方和平方根运算，梯度模值近似为：



$$G_M[f(x,y)] = |f(x,y) - f(x+1,y+1)| + |f(x+1,y) - f(x,y+1)| \quad (4-17)$$

利用双方向一次微分运算, 算出梯度后用梯度值替代该点的灰度值。在图像的最后一行或最后一列无法计算像素梯度时, 一般用前一行或前一列的梯度值近似代替。

为了在不破坏图像背景的前提下更好地增强边缘, 也可以对上述直接用梯度值代替灰度值的方法进行改进, 即利用门限判断来改进梯度锐化方法。具体公式如下:

$$G_M[f(x,y)] = \begin{cases} G_M[f(x,y)] + 100 & G[f(x,y)] \geq T \\ f(x,y) & \text{其他} \end{cases} \quad (4-18)$$

对图像而言, 物体和物体之间, 背景和背景之间的梯度变化一般很小, 灰度变化较大的地方一般集中在图像的边缘上, 也就是物体和背景交界的地方, 当我们设定一个合适的阈值  $T$ ,  $G_M[f(x,y)]$  大于等于  $T$  就认为该像素点处于图像的边缘, 对梯度值增加 100, 以使边缘变亮, 而对于  $G_M[f(x,y)]$  小于  $T$  就认为像素点是同类像素点 (同是背景或物体)。这样既增加了物体的边界, 又同时保留了图像背景原来的状态。

## 2. 拉普拉斯掩膜锐化

拉普拉斯算子是一种二阶导数算子, 是各向同性的微分运算, 与梯度算子一样, 具有旋转不变性, 可以满足不同走向的图像边界的锐化要求。

对一个连续函数  $f(x,y)$ , 它在点  $(x,y)$  处的拉普拉斯算子定义为

$$\nabla^2 f(x,y) = \frac{\partial^2 f}{\partial^2 x} + \frac{\partial^2 f}{\partial^2 y} \quad (4-19)$$

对数字图像来说, 离散函数  $f(x,y)$  的拉普拉斯算子可表示为:

$$\nabla^2 f(x,y) = [f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)] - 4f(x,y) \quad (4-20)$$

可以用图 4-2a 的模板形式表示。

对角线上的像素也可以加入到拉普拉斯变换中, 这样可将上式扩展为:

$$\nabla^2 f(x,y) = [f(x+1,y-1) + f(x+1,y+1) + f(x-1,y+1) + f(x-1,y-1) + f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1)] - 8f(x,y) \quad (4-21)$$

可以用图 4-2b 的模板形式表示。

0	1	0
1	-4	1
0	1	0

a) 拉普拉斯算子

1	1	1
1	-8	1
1	1	1

b) 扩展模板

图 4-2 拉普拉斯算子模板及其扩展模板



### 4.1.5 模糊复原技术

在照相的曝光期间，如果物体和相机有相对运动，则反映在底片上的图像有明显的移动，形成了模糊的运动图像。本节先分析图像只在  $X$  方向运动造成的模糊情况，在  $Y$  方向或者  $X$ 、 $Y$  方向均运动的情况可以类推。

设运动方向为  $X$  轴方向，则图像函数可以写成  $f(x-x_0(t), y)$ 。这里  $x_0(t)$  是沿  $X$  的正方向运动的距离。设曝光时间为  $0 < t < T$ ，在此期间移动的距离是  $a$ ，所以  $x_0(t) = at/T$ 。所形成的模糊图像为：

$$g(x, y) = a \int_0^T f(x - \frac{at}{T}, y) dt \quad (4-22)$$

式 (4-22) 中  $a$  是与照片感光灵敏度有关的系数，现已知  $g(x, y)$  和  $a$ ，求  $f(x, y)$ 。

由于模糊只在  $X$  轴方向，而和  $Y$  方向无关，所以可以一行一行地复原。在某一行  $y = y_0$  时，把  $g(x, y_0)$  和  $f(x, y_0)$  简写成  $g(x)$  和  $f(x)$ ，令  $\tau = x - \frac{at}{T}$ ，则式 4-22 可以简写为：

$$g(\tau) = \frac{1}{\beta} \int_{x-a}^x f(\tau) d\tau \quad (4-23)$$

对式 (4-23) 进行微分有：

$$f(x) = \beta g'(x) + f(x-a) \quad (4-24)$$

式中的  $\beta = \frac{a}{Ta}$ ，通过递推的方法可以求得原来图像的像素值。

设模糊图像的宽度为  $L$ ， $L = Ka$ 。并设  $x > L$  和  $x < 0$  范围的像素值为零。则变量  $x$  可以表示为：

$$x = z + ma \quad (4-25)$$

$z$  的取值范围在  $[0, a]$  之间， $m$  是  $\frac{x}{a}$  的整数部分，代入有：

$$f(z+ma) = \beta g'(z+ma) + f(z+ma-a) \quad (4-26)$$

设  $\varphi(z)$  为曝光期间在  $0 < x < a$  范围内移动的景物部分，即：

$$\varphi(z) = f(z-a) \quad (4-27)$$

通过  $\varphi(z)$ ，利用递推法可以求解。

当  $m=0$  时，有：

$$f(z) = \beta g'(z) + f(z-a) = g'(z) + \varphi(z) \quad (4-28)$$

当  $m=1$  时，有：

$$f(z+a) = \beta g'(z+a) + f(z) = \beta g'(z+a) + \beta g'(z) + \varphi(z) \quad (4-29)$$

以此类推，最后可以得到如下结果：

$$f(z+ma) = \sum_{k=0}^m \beta g'(z+ka) + \varphi(z) \quad 0 \leq z < a \quad (4-30)$$

$x = z + ma$  来代入中有:

$$f(x) = \sum_{k=0}^m \beta g'(x-ka) + \varphi(x-ma) \quad 0 \leq x < L \quad (4-31)$$

上式中  $\varphi(z)$  可由以下方法求出。

由于

$$\varphi(z) = -\sum_{k=0}^m \beta g'(z+ka) + f(z+ma) \quad (4-32)$$

式中  $m = 0, 1, \dots, k-1$ , 共有  $k$  项, 将上式以  $k$  项相加得:

$$k\varphi(z) = -\sum_{m=0}^{k-1} \sum_{k=0}^m \beta g'(z+ka) + \sum_{m=0}^{k-1} f(z+ma) \quad (4-33)$$

$$\varphi(z) = -\frac{1}{k} \sum_{m=0}^{k-1} \sum_{k=0}^m \beta g'(z+ka) + \frac{1}{k} \sum_{m=0}^{k-1} f(z+ma) \quad (4-34)$$

式 (4-34) 中第 2 项当  $k$  值较大时, 趋向于  $f$  的平均值, 因此可将它近似地看做一个常数, 所以设:

$$\frac{1}{k} \sum_{m=0}^{k-1} f(z+ma) = A \quad (4-35)$$

则有:

$$\varphi(z) = A - \frac{1}{k} \sum_{m=0}^{k-1} \sum_{k=0}^m \beta g'(z+ka) \quad (4-36)$$

最后可以得到:

$$f(x) = A + \sum_{k=0}^m \beta g'(x-ka) - \frac{\beta}{k} \sum_{m=0}^{k-1} \sum_{k=0}^m g'(x-ka) \quad (4-37)$$

#### 4.1.6 彩色增强技术

前面介绍的图像增强技术都是对灰度图像进行处理的, 而且结果也是灰度图像。彩色增强处理的对象也是灰度图像, 但生成的结果却是彩色图像。人的视觉对彩色相当敏感, 人眼一般能区分的灰度等级只有 20 多个, 但是能区分不同亮度、色度和饱和度的彩色却有几千种。根据人眼的这个特点, 可将彩色用于增强中, 以提高图像的可鉴别性。因此如果能将一幅灰度图像变成彩色图像, 就可以达到增强图像效果的目的。常用的彩色增强方法有真彩色增强、假彩色增强和伪彩色增强。

### 1. 真彩色增强

自然物体的彩色叫真彩色,把能真实反映自然物体本来颜色的图像叫真彩色图像。真彩色增强过程中,首先用加有红色滤片的黑白摄像机提取图像,将图像信号数字化后存储,然后分别用带有绿色滤片与蓝色滤片的摄像机获取图像并进行数字化后存储,最后就可以用彩色监视器将3幅图像合成一幅真彩色图像。

### 2. 假彩色增强

假彩色增强所处理的对象不是一幅灰度图像,而是一幅自然彩色图像或是同一景物的多光谱图像。通过映射函数变换成新的三基色分量,彩色合成使增强后的图像中各目标呈现出与原图像中不同的彩色,这种技术称为假彩色增强。假彩色增强的目的有2个:一是使感兴趣的目标呈现奇异的彩色或置于奇特的彩色环境中,从而更受人注目;另一个是使景物呈现出与人眼色觉相匹配的颜色,以提高对目标的分辨力。多光谱图像的假彩色增强可表示为:

$$R(x,y) = F_R[f_1(x,y), f_2(x,y) \cdots f_N(x,y)]$$

$$G(x,y) = F_G[f_1(x,y), f_2(x,y) \cdots f_N(x,y)]$$

$$B(x,y) = F_B[f_1(x,y), f_2(x,y) \cdots f_N(x,y)]$$

式中,  $F_R$ 、 $F_G$ 、 $F_B$  为映射函数,  $R(x,y)$ 、 $G(x,y)$  和  $B(x,y)$  为显示空间三基色分量。

### 3. 伪彩色增强

伪彩色增强是把灰度图像的各个不同灰度级按照线性或者非线性的映射函数变换成不同的彩色,使图像细节更容易辨认,目标更容易识别。

可以有多种方式实现从灰度空间到彩色空间的变换,最简单的就是把灰度图像的灰度级从0~255分成255个区间,给每个区间指定一种彩色。这种方法比较简单,但是变换出的彩色有限。

从灰度空间到彩色空间的一种较有代表性的变换方式是将原图像  $f(x,y)$  的灰度经过红、绿、蓝3种不同变换,变成三基色的分量  $R(x,y)$ 、 $G(x,y)$ 、 $B(x,y)$ ,生成相应的彩色。彩色的含量由变换函数而定。

还有一种伪彩色增强是在频域内实现的。先将图像转换到频域,然后在频域用不同特性的滤波器分离成3个独立的分量,再进行傅里叶逆变换得到3幅代表不同频率分量的单色图像,再对这3幅图像进行进一步处理,并合成在一起,从而实现了彩色增强。

## 4.1.7 滤镜技术

滤镜是一种改变图像外观的程序,其本身并不属于数字图像处理研究的范畴,但是可以认为滤镜是数字图像处理的一个应用。如果没有相关的图像处理理论指导,这些滤镜效果都是不可能实现的。滤镜程序的核心算法都源自数字图像处理知识,唯有对数字图像处理有深刻的理解才能设计实现滤镜。

### 1. 底片效果

传统相机的成像过程一般是这样的：物像的反射光经过镜头聚焦在胶片上，胶片上的感光剂随光发生变化，即胶片通过一次性反色曝光成形。冲洗相片时，则将胶片上的物像在暗室中反曝在相纸上，用显影液浸泡显像，再用定影液浸泡，晾干即可。而本节我们要讲的是通过数字图像处理的手段将一张照片还原回底片效果，这种滤镜就是“底片效果”。实现底片效果的滤镜可以将一张照片还原回其相应的底片效果。

图像反色的原理实际上就是取每一个像素点的相对立颜色值。在一个 RGB 色彩空间中，任何一种颜色都对应着笛卡儿坐标系中的一个实点，而其相对应的反色也就是空间中心对称点。例如图像中某一点的 RGB 色彩值为 RGB (50, 100, 200)，那么它所对应的反色值就是 RGB (205, 155, 55)。图 4-3 为 Lena 图像底片的效果演示。

### 2. 雕刻效果

“雕刻效果”，顾名思义就是使图像产生雕刻感，雕刻效果使图像的前景前向凸出，常见于一些纪念碑的雕刻上。“雕刻效果”是增强图像立体感的有效手段。

要实现图像的雕刻化处理非常简单，只需把图像上的一个像素和其右下  $45^\circ$  方向上最临近的像素进行求差运算。为了保证图像能够呈现出灰色，处理过程中还需为这个差值加上一个常数 128，然后将其作为新的像素值即可。需要注意的是最终颜色分量的取值范围应介于 0~255，如果超出范围应做相应的处理。此外，考虑到图像的底边及右侧的两组像素无法取得新像素值，故将其置为灰色。当设置一个像素值时，它和它右下方的像素都要被用到，为了避免用到已经设置过的像素，应该从图像的左上方像素开始处理。图 4-4 为 Lena 图像雕刻效果的演示。



图 4-3 Lena 图像底片的效果演示



图 4-4 Lena 图像雕刻的效果演示

### 3. 黑白效果

黑白效果的照片在带给人们无限的怀旧感的同时，也带来了一种艺术的美感。从实质上分析，

如果场景已经平淡无奇,那么完全去除色彩的照片将使沉闷的景象变得令人感兴趣。黑白相片可传达一种年代久远的情感,这是彩色相片所无法做到的。所谓照片只是本体物像的二维表示。如果场景中的颜色转换为灰度级,从漆黑色转为亮白色,则相片的其他方面,如形态、亮度、对比度、纹理和色调等元素就将占据主导地位,于是黑白照片除了给人以怀旧感之外,在辅助分析中也有着十分重要的作用。

RGB 颜色模型对应笛卡儿坐标系中的一个立方体, R、G、B 分别代表 3 个坐标轴。当 R、G、B 对应的数值都取 0 时,即表示坐标原点处,表示黑色;反之当 R、G、B 对应的数值都取最大值时则表示白色。立方体空间中的其他各点表示其他颜色。立方体的主对角线,即由原点到 RGB 取最大值的坐标点之间的连线,表示色彩的灰度级。也就是说色彩沿这条线的方向将在黑白之间变化,这也就揭示了显示灰度图像的原理,即当 RGB 三者的取值相同时,这个色彩将表示黑白之间的一个灰度值,因为主对角线上的任意一点 RGB 值都相等。

下面简单介绍一下彩色图像转化为黑白图像的方法:

- 最大值法,最大值法使每个像素点的 RGB 值都等于原来像素点中 RGB 值的最值,即  $\text{MAX}(R, G, B)$ 。
- 平均值法,相对于最大值法,平均值法将会形成比较柔和的黑白图像。其具体做法是使每个像素点的 RGB 值等于原像素点的 RGB 值的平均值,即  $R=G=B=(R+G+B)/3$ 。
- 加权平均值法,加权平均值法会根据指定的每个像素点的 RGB 分量的权值,并取其加权平均值而得到,即  $R=G=B=(R*x+G*y+B*z)/3$ 。其中,  $x$  代表该像素点 R 分量的权数,  $y$  代表该像素点 G 分量的权数,  $z$  代表该像素点 B 分量的权数,且  $x, y, z > 0$ 。

图 4-5 为 cat 图像黑白效果的演示。



a) 彩色图像

b) 黑白图像

图 4-5 cat 图像黑白效果的演示

#### 4. 雾化效果

雾化效果,顾名思义,就是一种模糊的效果,就像我们要看的物体在雾中一样,在滤镜设计

上恰当地应用随机化处理将产生意想不到的效果。

雾化处理的实现方法基于一定的随机机制，这里将讨论3种雾化处理的方式。

- 水平方式，水平雾化将某点的像素值改变为水平方向上另一个随机点的像素值，随机点的选取由随机函数控制。通过对随机范围的控制可以调整雾化的程度，也就是说当随机空间变大时，置换原像素点的像素越远可能性就越大，处理的现象也就越明显。
- 垂直方式，垂直雾化将某点的像素值改变为垂直方向上另一个随机点的像素值，随机点的选取由随机函数控制。通过对随机范围的控制可以调控雾化的程度。垂直雾化由于进行了垂直方向上的像素置换而使图像呈现出垂直方向上的毛躁感。
- 复合方式，复合雾化其实就是水平雾化和垂直雾化的组合，它将更接近物像的本体。复合雾化的具体实现方法是将某点的像素值改变为斜向  $45^\circ$  方向上另一个随机点的像素值，随机点的选取由随机函数控制。

图4-6为Lena图像雾化效果的演示。



图4-6 Lena图像雾化效果的演示



## 5. 素描效果

素描是美术技法中十分基础的一种，它是绘画者在既定面积或在物质的平面上，描绘出外在的形体及表现物体在空间中位置的一种技法。素描的特点是用线与面的表现方式来表达物体的形象，它用明暗层次来表现一个物体在光照下的效果，因此素描可以帮助绘画者掌握物体的明暗层次和基本形象。实现素描效果需要解决的关键问题主要集中在两个方面，首先是准确地描绘图像的轮廓，其次是准确地反映物体的光照情况。

实现图像素描效果的原理比较复杂，它应用了数字图像处理理论中非常重要的理论——边缘检测。简单地说主要分为以下几个步骤：首先借助拉普拉斯算子生成素描图，处理结果图可以较好地描述图形的轮廓线条，但其不足在于结果包含了过多的细节信息而易受各类杂点影响。考虑到以上条件，为了得到较理想的素描图像，这里的做法是先设计合适的模板来对图像进行降噪，然后再对素描图进行模糊处理，最终将得到柔和的素描线条。

图 4-7 为 Lena 图像素描效果的演示。



图 4-7 Lena 图像素描效果的演示

## 4.2 系统功能

Photoshop 是一款久负盛名的图像处理软件，集图像扫描、编辑修改、图像制作、广告创意、图像输出等功能于一身，深受广大平面设计人员和电脑美术爱好者的喜爱。本章讲解的图像编辑器模拟实现 Photoshop 软件的部分图像编辑功能。

### 4.2.1 功能描述

图像编辑器实现的主要功能有：

- 1) 灰度变换增强功能。主要有线性灰度增强、分段线性增强和对数非线性增强等。



- 2) 直方图增强功能。主要有直方图均衡化和直方图规定化两个功能。
- 3) 平滑去噪功能。主要有邻域平均平滑、加权平均平滑、选择式掩膜平滑和中值滤波法这几种功能。
- 4) 图像锐化功能。主要有门限梯度锐化和拉普拉斯锐化两种方法来实现该功能。
- 5) 彩色增强功能。本系统主要实现了增强彩色的功能。
- 6) 模糊复原功能。将模糊的运动图像进行复原的功能。
- 7) 滤镜效果功能。主要实现底片效果、雕刻效果、黑白效果等功能。

### 4.2.2 界面效果

图像编辑器运行的界面效果如图 4-8 所示, 图中显示的是对图像进行拉普拉斯锐化后的效果。



图 4-8 图像编辑器运行效果

## 4.3 系统结构与流程

图像编辑器主要包括灰度变换增强、直方图增强、平滑去噪、图像锐化、彩色增强、模糊复原和滤镜效果 7 大模块组成。

### 4.3.1 总体结构

图像编辑器主要有 7 大模块, 每个模块下又有一些不同的小模块, 具体系统结构如图 4-9 所示。

### 4.3.2 主要流程

图像编辑器系统的主要流程如图 4-10 所示。

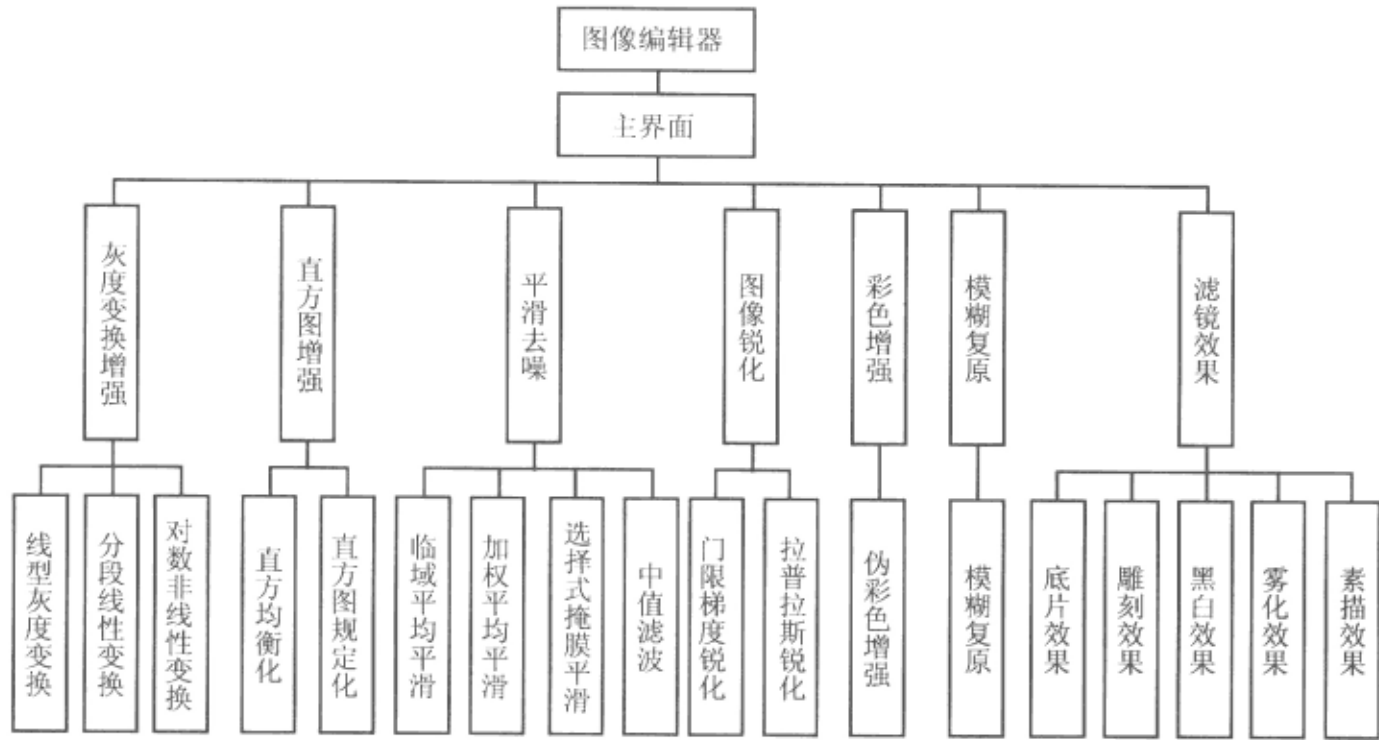


图 4-9 图像编辑器系统结构图

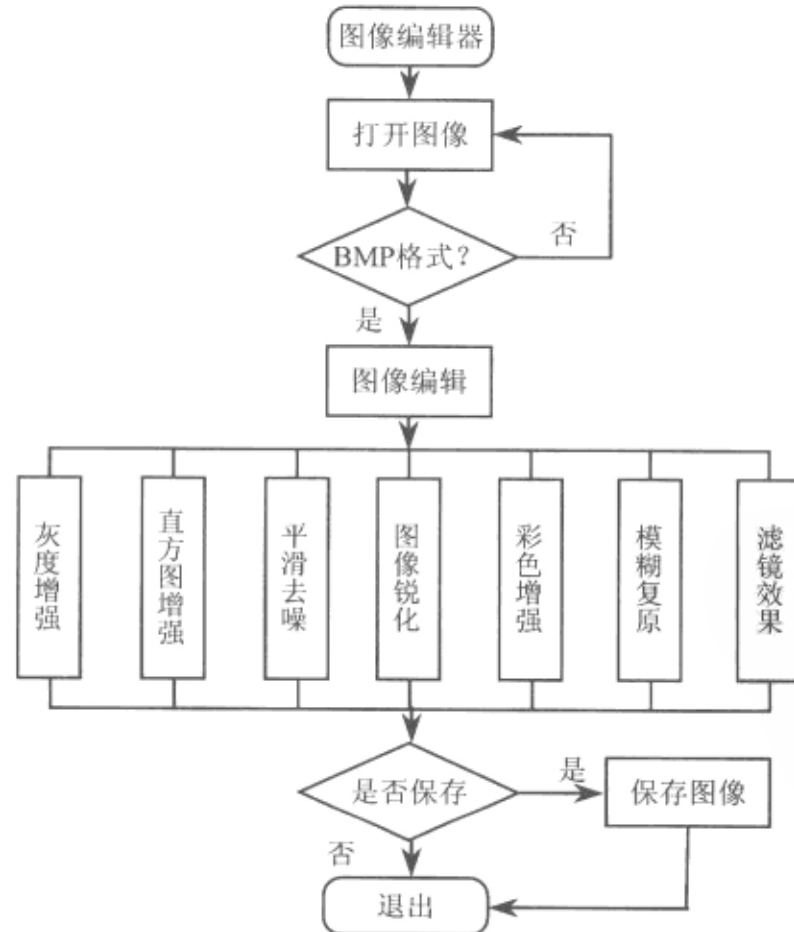


图 4-10 图像编辑器的主要流程

## 4.4 编程实现

图像编辑器采用 VC 2008 开发平台编程实现。

### 4.4.1 灰度变换增强模块

灰度变换主要针对独立的像素点进行处理,由输入像素点的灰度值来决定相应的输出像素点的灰度值,通过改变原始图像数据所占据的灰度范围而使图像在视觉上得到改观。灰度增强方法又分为线性灰度增强、分段线性灰度增强与非线性灰度增强,下面分别介绍一下各个模块的编程实现。

图像的线性灰度变换算法的程序代码如下:

```

BOOL CGrayTransformDib::Linear_Transform(BYTE gMin, BYTE gMax)
{
    LPBYTE lpSrc;           // 指向原图像的指针
    LPBYTE lpDst;           // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;    // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i, j;              // 循环变量
    BYTE pixel;             // 像素值
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();    // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight();  // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存,以保存新图像
    if (hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    // 初始化新分配的内存,设定初始值为 0
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight);
    // 逐个扫描图像中的像素点,进行灰度线性变换
    for(j = 0; j <lHeight; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            // 指向原图像倒数第 j 行,第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
            // 指向目标图像倒数第 j 行,第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            // 取得当前指针处的像素值,注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;

```

```

        //根据公式(4-2) 求出目标图像中与当前点对应的像素点的灰度值
        *lpDst = (BYTE)((float)(gMax-gMin)/255)*pixel+gMin+0.5);
    }
}
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);    // 复制变换后的图像
//释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

图像的分段线性灰度变换增强算法的程序代码如下:

```

BOOL CGrayTransformDib::Seg_Linear_Transform(BYTE gSrc1,
BYTE gSrc2,BYTE gDst1, BYTE gDst2)
{
    LPBYTE lpSrc;                // 指向原图像的指针
    LPBYTE lpDst;                // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;         // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j;                    //循环变量
    BYTE pixel;                  //像素值
    LPBYTE lpDIBBits=m_pDib->GetData(); //找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); //获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); //获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); //暂时分配内存以保存新图像
    if (hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight); // 初始化新分配的内存, 设定初始值为 0
    //逐个扫描图像中的像素点, 进行灰度分段线性变换
    for(j = 0; j <lHeight; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            // 指向原图像倒数第 j 行, 第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
            // 指向目标图像倒数第 j 行, 第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            //取得当前指针处的像素值, 注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;
            //根据公式(4-3) 右边上式求出目标图像中与当前点对应的像素点的灰度值
            if(pixel<gSrc1)
            {

```

```

        *lpDst = (BYTE)((float)gDst1/gSrc1)*pixel+0.5);
    }
    //根据公式(4-3)右边中式求出目标图像中与当前点对应的像素点的灰度值
    if((pixel>gSrc1)&&(pixel<=gSrc2))
    {
        *lpDst = (BYTE)((float)(gDst2-gDst1)/(gSrc2-gSrc1))*(pixel-gSrc1)
+gDst1+0.5);
    }
    //根据公式(4-3)右边下式求出目标图像中与当前点对应的像素点的灰度值
    if((pixel>gSrc2)&&(pixel<=255))
    {
        *lpDst=(BYTE)((float)(255-gDst2)/(255-gSrc2))*(pixel-gSrc2)
+gDst2+0.5);
    }
}
}
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
//释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

图像的对数函数非线性变换算法实现的程序代码如下:

```

BOOL CGrayTransformDib::Log_Transform(double a, double b, double c)
{
    LPBYTE lpSrc; // 指向原图像的指针
    LPBYTE lpDst; // 指向缓存图像的指针
    LPBYTE lpNewDIBBits; // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j; // 循环变量
    BYTE pixel; // 像素值
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存以保存新图像
    if (hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    // 初始化新分配的内存, 设定初始值为 0
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight);
    // 逐个扫描图像中的像素点, 进行对数函数非线性灰度变换
    for(j = 0; j <lHeight; j++)

```

```

    {
        for(i = 0; i < lWidth; i++)
        {
            // 指向原图像倒数第 j 行, 第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
            // 指向目标图像倒数第 j 行, 第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            //取得当前指针处的像素值, 注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;
            //根据公式(4-4) 求出目标图像中与当前点对应的像素点的灰度值
            *lpDst = (BYTE)((log((double)(pixel+1)))/(b*log(c))+a+0.5);
        }
    }
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
    //释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);
    return TRUE;
}

```

#### 4.4.2 直方图增强模块

直方图增强模块主要由直方图统计算法、直方图均衡化和直方图规定化三部分组成, 各部分的具体编程实现如下所述。

图像的灰度直方图统计算法实现的程序代码如下:

```

void CHistogramDib::Histogram_Statistic( float *probability)
{
    LPBYTE lpSrc;           //指向原图像的指针
    long i, j;              //循环变量
    int gray[256];          //灰度计数
    BYTE pixel;             //像素值
    LPBYTE lpDIBBits=m_pDib->GetData(); //找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();    //获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight();  //获得原图像的高度
    memset(gray, 0, sizeof(gray));     // 灰度计数变量初始化
    //逐个扫描图像中的像素点, 进行灰度计数统计
    for(j = 0; j < lHeight; j++)
    {
        for(i = 0; i < lWidth; i++)
        {
            // 指向原图像倒数第 j 行, 第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
            //取得当前指针处的像素值, 注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;
            gray[pixel]++; // 灰度统计计数
        }
    }
}

```

## 第4章

Visual C++

```

    }
}
// 计算灰度概率密度
for(i=0;i<256;i++)
{
    probability[i] = gray[i]/(lHeight * lWidth *1.0f);
}
}

```

图像的灰度直方图均衡化算法实现的程序代码如下:

```

BOOL CHistogramDib::Histogram_Equalization( )
{
    LPBYTE lpSrc;           // 指向原图像的指针
    LPBYTE lpDst;           // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;    // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j;               // 循环变量
    BYTE pixel;             // 像素值
    float fPro[256];        // 原图像灰度分布概率密度变量
    float temp[256];        // 中间变量
    int nRst[256];          // 中间变量
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();    // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight();   // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存保存新图像
    if(hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight); // 初始化新分配的内存, 设定初始值为 0
    memset(temp, 0, sizeof(temp));           // 初始化中间变量 temp
    Histogram_Statistic(fPro);                // 获取原图像灰度分布的概率密度
    // 进行直方图均衡化处理
    for(i = 0; i < 256; i++)
    {
        if(i == 0)
        {
            temp[0] = fPro[0];
        }
        else
        {
            temp[i] = temp[i-1] + fPro[i];
        }
        nRst[i] = (int)(255.0f * temp[i] + 0.5f);
    }
}

```



```

    }
    //将直方图均衡化后的结果写到目标图像中
    for(j = 0; j < lHeight; j++)
    {
        for(i = 0; i < lWidth; i++)
        {
            // 指向原图像倒数第 j 行, 第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
            // 指向目标图像倒数第 j 行, 第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            //取得当前指针处的像素值, 注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;
            *lpDst = (BYTE)(nRst[pixel]);
        }
    }
    // 复制均衡化处理后的图像到原图像中
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);
    //释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);
    return TRUE;
}

```

图像的直方图规定化算法实现的程序代码如下:

```

BOOL CHistogramDib::Histogram_Match(BYTE bGray, int *npMap, float *fpPro)
{
    LPBYTE lpSrc;           // 指向原图像的指针
    LPBYTE lpDst;           // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;    // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i, j;              // 循环变量
    BYTE pixel;             // 像素值
    float fPro[256];        // 原图像灰度分布概率密度变量
    float temp[256];        // 中间变量
    int nMap[256];          // 灰度映射表变量
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();    // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight();   // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存保存新图像
    if(hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE)LocalLock(hNewDIBBits);
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight); // 初始化新分配的内存设定初始值为 0
}

```

```

Histogram_Statistic(fPro);                                // 获取原图像灰度分布的概率密度
// 计算原图像累计直方图
for (i = 0; i < 256; i++)
{
    if (i == 0)
    {
        temp[0] = fPro[0];
    }
    else
    {
        temp[i] = temp[i-1] + fPro[i];
    }
    fPro[i] = temp[i];
}
// 计算规定变换后的累计直方图
for (i = 0; i < bGray; i++)
{
    if (i == 0)
    {
        temp[0] = fpPro[0];
    }
    else
    {
        temp[i] = temp[i-1] + fpPro[i];
    }
    fpPro[i] = temp[i];
}
// 确定映射关系
for (i = 0; i < 256; i++)
{
    int m = 0;                                           // 最接近的规定直方图灰度级变量
    float min_value = 1.0f;                             // 最小差值变量
    // 枚举规定直方图各灰度
    for (j = 0; j < bGray; j++)
    {
        float now_value = 0.0f;                        // 当前差值变量
        // 差值计算
        if (fPro[i] - fpPro[j] >= 0.0f)
            now_value = fPro[i] - fpPro[j];
        else
            now_value = fpPro[j] - fPro[i];
        // 寻找最接近的规定直方图灰度级
        if (now_value < min_value)
        {
            m = j;                                       // 最接近的灰度级
            min_value = now_value;                     // 最小差值
        }
    }
}

```

## 数字图像处理典型案例详解

```

    }
    nMap[i] = npMap[m];           // 建立灰度映射表
}
// 对各像素进行直方图规定化映射处理
for (j = 0; j < lHeight; j++)
{
    for (i = 0; i < lWidth; i++)
    {
        // 指向原图像倒数第 j 行, 第 i 个像素的指针
        lpSrc = (LPBYTE)lpDIBBits + lWidth * j + i;
        // 指向目标图像倒数第 j 行, 第 i 个像素的指针
        lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
        // 取得当前指针处的像素值, 注意要转换为 BYTE 类型
        pixel = (BYTE)*lpSrc;
        *lpDst = (BYTE)(nMap[pixel]); // 对目标图像进行映射处理
    }
}
// 复制直方图规定化处理后的图像到原图像中
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);
// 释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

## 4.4.3 平滑去噪模块

平滑去噪模块主要分为邻域平均平滑子模块、加权平均平滑子模块、选择式掩膜平滑和中值滤波 4 个子模块, 各子模块的具体编程实现如下:

灰度图像的邻域平均平滑算法实现的程序代码如下:

```

BOOL CSmoothProcessDib::Average_Smooth( )
{
    LPBYTE lpDst;           // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;    // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i, j;              // 循环变量
    BYTE average;           // 邻域均值变量

    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();      // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight();    // 获得原图像的高度
    // 暂时分配内存, 以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);
    if (hNewDIBBits == NULL)
    {

```

```

        return FALSE;
    }
    lpNewDIBBits = (LPBYTE)LocalLock(hNewDIBBits);
    // 初始化新分配的内存, 设定初始值为 0
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight);
    // 逐个扫描图像中的像素点, 求其邻域均值
    for(j = 1; j < lHeight-1; j++)
    {
        for(i = 1; i < lWidth-1; i++)
        {
            // 指向目标图像第 j 行, 第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            // 求当前点及其周围 8 个点的均值
            average=(BYTE)((float)(lpDIBBits[(j-1)*lWidth+(i-1)]+lpDIBBits
[(j-1)*lWidth+i] +lpDIBBits[(j-1)*lWidth+(i+1)]+lpDIBBits[j*lWidth+(i-1)]
+lpDIBBits[j*lWidth+i]+lpDIBBits[j*lWidth+i+1]
+lpDIBBits[(j+1)*lWidth+(i-1)]+lpDIBBits[(j+1)*lWidth+i]
+lpDIBBits[(j+1)*lWidth+i+1])/9+0.5);
            *lpDst = average; // 将求得的均值赋值给目标图像中与当前点对应的像素点
        }
    }
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);
    return TRUE;
}

```

灰度图像的加权平均平滑算法实现的程序代码如下:

```

BOOL CSmoothProcessDib::Value_Average_Smooth(int Structure[3][3])
{
    LPBYTE    lpDst;                // 指向缓存图像的指针
    LPBYTE    lpNewDIBBits;         // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j,m,n;                  // 循环变量
    int sum=0;                      // 模板中各个元素总和
    BYTE value_average;             // 领域加权均值变量
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth();   // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); // 获得原图像的高度
    // 暂时分配内存, 以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);
}

```

```

if(hNewDIBBits == NULL)
{
    return FALSE;
}
lpNewDIBBits = (LPBYTE)LocalLock(hNewDIBBits);
lpDst = (LPBYTE)lpNewDIBBits;
memset(lpDst, (BYTE)0, lWidth * lHeight); // 初始化新分配的内存设定初始值为 0
//求模板中各元素权值总和
for (m = 0;m < 3;m++)
{
    for (n = 0;n < 3;n++)
    {
        sum+=Structure[m][n];
    }
}
//逐个扫描图像中的像素点, 求其邻域加权均值
for(j = 1; j < lHeight-1; j++)
{
    for(i = 1;i < lWidth-1; i++)
    {
        // 指向目标图像第 j 行, 第 i 个像素的指针
        lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
        //求加权均值
        value_average=(BYTE)((float) (lpDIBBits[(j+1)*lWidth+(i-1)]
            *Structure[0][0]
            +lpDIBBits[(j+1)*lWidth+i]*Structure[0][1]
            +lpDIBBits[(j+1)*lWidth+i+1]*Structure[0][2]
            +lpDIBBits[j*lWidth+(i-1)]*Structure[1][0]
            +lpDIBBits[j*lWidth+i]*Structure[1][1]
            +lpDIBBits[j*lWidth+i+1]*Structure[1][2]
            +lpDIBBits[(j-1)*lWidth+(i-1)]*Structure[2][0]
            +lpDIBBits[(j-1)*lWidth+i]*Structure[2][1]
            +lpDIBBits[(j-1)*lWidth+(i+1)]
            *Structure[2][2]) /sum+0.5);
        //将求得的加权均值赋值给目标图像中与当前点对应的像素点
        *lpDst = value_average;
    }
}
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
//释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

灰度图像的选择式掩膜平滑处理的程序代码如下:

```

BOOL CSmoothProcessDib::Select_Smooth( )

```

```

{
    LPBYTE lpDst;                // 指向缓存图像的指针
    LPBYTE lpNewDIBBits;        // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    int i,j,n;                  // 循环变量
    BYTE pixel[9];              // 邻域各点的像素值
    float mean[9],var[9],varMin; // 邻域均值, 邻域方差, 方差最小值
    int nMin;                   // 方差最小时的邻域号
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存保存新图像
    if (hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    // 初始化新分配的内存, 设定初始值为 0
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight);
    // 求 9 种邻域的均值与方差
    for(j=2;j<=lHeight-3;j++)
    {
        for(i=2;i<=lWidth-3;i++)
        {
            // 第 1 邻域
            pixel[0]=lpDIBBits[(j-1)*lWidth+(i-1)];
            pixel[1]=lpDIBBits[(j-1)*lWidth+i];
            pixel[2]=lpDIBBits[(j-1)*lWidth+(i+1)];
            pixel[3]=lpDIBBits[j*lWidth+(i-1)];
            pixel[4]=lpDIBBits[j*lWidth+i];
            pixel[5]=lpDIBBits[j*lWidth+(i+1)];
            pixel[6]=lpDIBBits[(j+1)*lWidth+(i-1)];
            pixel[7]=lpDIBBits[(j+1)*lWidth+i];
            pixel[8]=lpDIBBits[(j+1)*lWidth+(i+1)];
            mean[0]=(float) (pixel[0]+pixel[1]+pixel[2]+pixel[3]
                             +pixel[4]+pixel[5]
                             +pixel[6]+pixel[7]+pixel[8])/9;
            var[0]=0;
            for(n=0;n<=8;n++)
                var[0]+=pixel[n]*pixel[n]-mean[0]*mean[0];
            // 第 2 邻域
            pixel[0]=lpDIBBits[(j-2)*lWidth+(i-1)];
            pixel[1]=lpDIBBits[(j-2)*lWidth+i];
            pixel[2]=lpDIBBits[(j-2)*lWidth+(i+1)];
            pixel[3]=lpDIBBits[(j-1)*lWidth+(i-1)];
            pixel[4]=lpDIBBits[(j-1)*lWidth+i];

```

```

pixel[5]=lpDIBBits[(j-1)*lWidth+(i+1)];
pixel[6]=lpDIBBits[j*lWidth+i];
        mean[1]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]
                        +pixel[4]
                        +pixel[5]+pixel[6])/7;

var[1]=0;
for(n=0;n<=6;n++)
    var[1]+=pixel[n]*pixel[n]-mean[1]*mean[1];
    //第3邻域
pixel[0]=lpDIBBits[(j-1)*lWidth+(i-2)];
pixel[1]=lpDIBBits[(j-1)*lWidth+(i-1)];
pixel[2]=lpDIBBits[j*lWidth+(i-2)];
pixel[3]=lpDIBBits[j*lWidth+(i-1)];
pixel[4]=lpDIBBits[j*lWidth+i];
pixel[5]=lpDIBBits[(j+1)*lWidth+(i-2)];
pixel[6]=lpDIBBits[(j+1)*lWidth+(i-1)];
mean[2]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[2]=0;
for(n=0;n<=6;n++)
    var[2]+=pixel[n]*pixel[n]-mean[2]*mean[2];
    //第4邻域
pixel[0]=lpDIBBits[j*lWidth+i];
pixel[1]=lpDIBBits[(j+1)*lWidth+(i-1)];
pixel[2]=lpDIBBits[(j+1)*lWidth+i];
pixel[3]=lpDIBBits[(j+1)*lWidth+(i+1)];
pixel[4]=lpDIBBits[(j+2)*lWidth+(i-1)];
pixel[5]=lpDIBBits[(j+2)*lWidth+i];
pixel[6]=lpDIBBits[(j+2)*lWidth+(i+1)];

mean[3]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[3]=0;
for(n=0;n<=6;n++)
    var[3]+=pixel[n]*pixel[n]-mean[3]*mean[3];
    //第5邻域
pixel[0]=lpDIBBits[(j-1)*lWidth+(i+1)];
pixel[1]=lpDIBBits[(j-1)*lWidth+(i+2)];
pixel[2]=lpDIBBits[j*lWidth+i];
pixel[3]=lpDIBBits[j*lWidth+(i+1)];
pixel[4]=lpDIBBits[j*lWidth+(i+2)];
pixel[5]=lpDIBBits[(j+1)*lWidth+(i+1)];
pixel[6]=lpDIBBits[(j+1)*lWidth+(i+2)];
mean[4]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[4]=0;
for(n=0;n<=6;n++)

```



```

        var[4]+=pixel[n]*pixel[n]-mean[4]*mean[4];
//第6邻域
pixel[0]=lpDIBBits[(j-2)*lWidth+(i+1)];
pixel[1]=lpDIBBits[(j-2)*lWidth+(i+2)];
pixel[2]=lpDIBBits[(j-1)*lWidth+i];
pixel[3]=lpDIBBits[(j-1)*lWidth+(i+1)];
pixel[4]=lpDIBBits[(j-1)*lWidth+(i+2)];
pixel[5]=lpDIBBits[j*lWidth+i];
pixel[6]=lpDIBBits[j*lWidth+(i+1)];
mean[5]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[5]=0;
for(n=0;n<=6;n++)
    var[5]+=pixel[n]*pixel[n]-mean[5]*mean[5];
//第7邻域
pixel[0]=lpDIBBits[(j-2)*lWidth+(i-2)];
pixel[1]=lpDIBBits[(j-2)*lWidth+(i-1)];
pixel[2]=lpDIBBits[(j-1)*lWidth+(i-2)];
pixel[3]=lpDIBBits[(j-1)*lWidth+(i-1)];
pixel[4]=lpDIBBits[(j-1)*lWidth+i];
pixel[5]=lpDIBBits[j*lWidth+(i-1)];
pixel[6]=lpDIBBits[j*lWidth+i];
mean[6]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[6]=0;
for(n=0;n<=6;n++)
    var[6]+=pixel[n]*pixel[n]-mean[6]*mean[6];
//第8邻域
pixel[0]=lpDIBBits[j*lWidth+(i-1)];
pixel[1]=lpDIBBits[j*lWidth+i];
pixel[2]=lpDIBBits[(j+1)*lWidth+(i-2)];
pixel[3]=lpDIBBits[(j+1)*lWidth+(i-1)];
pixel[4]=lpDIBBits[(j+1)*lWidth+i];
pixel[5]=lpDIBBits[(j+2)*lWidth+(i-2)];
pixel[6]=lpDIBBits[(j+2)*lWidth+(i-1)];
mean[7]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                +pixel[5]+pixel[6])/7;

var[7]=0;
for(n=0;n<=6;n++)
    var[7]+=pixel[n]*pixel[n]-mean[7]*mean[7];
//第9邻域
pixel[0]=lpDIBBits[j*lWidth+i];
pixel[1]=lpDIBBits[j*lWidth+(i+1)];
pixel[2]=lpDIBBits[(j+1)*lWidth+i];
pixel[3]=lpDIBBits[(j+1)*lWidth+(i+1)];
pixel[4]=lpDIBBits[(j+1)*lWidth+(i+2)];
pixel[5]=lpDIBBits[(j+2)*lWidth+(i+1)];

```

## 数字图像处理典型案例详解

```

        pixel[6]=lpDIBBits[(j+2)*lWidth+(i+2)];
        mean[8]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+pixel[4]
                        +pixel[5]+pixel[6])/7;

        var[8]=0;
        for(n=0;n<=6;n++)
            var[8]+=pixel[n]*pixel[n]-mean[8]*mean[8];
        //求方差最小的邻域nMin
        varMin=var[0];
        nMin=0;
        for(n=0;n<=8;n++)
        {
            if(varMin>var[n])
            {
                varMin=var[n];
                nMin=n;
            }
        }

        // 指向目标图像第 j 行, 第 i 个像素的指针
        lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
        //将方差最小的邻域均值赋值给目标像素点
        *lpDst = (BYTE)(mean[nMin]+0.5);
    }
}

memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
//释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

灰度图像的中值滤波算法实现的程序代码如下:

```

BOOL CSmoothProcessDib::Middle_Smooth( )
{
    LPBYTE lpDst; // 指向缓存图像的指针
    LPBYTE lpNewDIBBits; // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    int i,j,x,y,m; // 循环变量
    int flag; // 循环标志变量
    BYTE pixel[9],mid; // 窗口像素值及中值
    BYTE temp; // 中间变量
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存, 以保存新图像
    if (hNewDIBBits == NULL)
    {

```

```

        return FALSE;
    }
    lpNewDIBBits = (LPBYTE)LocalLock(hNewDIBBits);
    lpDst = (LPBYTE)lpNewDIBBits;
    memset(lpDst, (BYTE)0, lWidth * lHeight); // 初始化新分配的内存设定初始值为 0
    //中值滤波
    for(j=1;j<lHeight-1;j++)
    {
        for(i=1;i<lWidth-1;i++)
        {
            //把 3*3 屏蔽窗口的所有像素值放入 pixel[m]
            m=0;
            for(y=j-1;y<=j+1;y++)
            for(x=i-1;x<=i+1;x++)
            {
                pixel[m]=lpDIBBits[y*lWidth+x];
                m++;
            }
            //把 pixel[m] 中的值按降序排序
            do{
                flag=0;
                for(m=0;m<9;m++)
                {
                    if(pixel[m]<pixel[m+1])
                    {
                        temp=pixel[m];
                        pixel[m]=pixel[m+1];
                        pixel[m+1]=temp;
                        flag=1;
                    }
                }
            }while(flag==1);
            mid=pixel[4]; //求中值 mid
            // 指向目标图像第 j 行, 第 i 个像素的指针
            lpDst = (LPBYTE)lpNewDIBBits + lWidth * j + i;
            *lpDst = mid; //将中值赋给目标图像的当前点
        }
    }
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); //复制变换后的图像
    //释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);
    return TRUE;
}

```

#### 4.4.4 图像锐化模块

图像锐化模块包括门限梯度锐化和拉普拉斯锐化两个子模块。灰度图像的门限梯度锐化算法实现的程序代码如下：

```

BOOL CSharpenProcessDib::GateGrad(BYTE t)
{
    LPBYTE lpNewDIBBits;           // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j;                      // 循环变量
    BYTE temp;                     // 暂存双向一次微分结果
    LPBYTE lpDIBBits=m_pDib->GetData(); // 找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); // 获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); // 获得原图像的高度
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight); // 暂时分配内存以保存新图像
    if (hNewDIBBits == NULL)
    {
        return FALSE;
    }
    lpNewDIBBits = (LPBYTE )LocalLock(hNewDIBBits);
    // 初始化新分配的内存, 设定初始值为 0
    memset(lpNewDIBBits, (BYTE)0, lWidth * lHeight);
    // 逐个扫描图像中的像素点, 进行门限梯度锐化处理
    for(j=1;j<lHeight-1;j++)
    {
        for(i=1;i<lWidth-1;i++)
        {
            // 根据双向一次微分公式计算当前像素的灰度值
            temp=(BYTE)sqrt((float)((lpDIBBits[lWidth*j+i]-lpDIBBits[lWidth*j+(i-1)])
                                   *(lpDIBBits[lWidth*j+i]-lpDIBBits[lWidth*j+(i+1)])
                                   +(lpDIBBits[lWidth*j+i]-lpDIBBits[lWidth*(j-1)+i])
                                   *(lpDIBBits[lWidth*j+i]-lpDIBBits[lWidth*(j-1)+i])));
            if (temp>=t)
            {
                if((temp+100)>255)
                    lpNewDIBBits[lWidth*j+i]=255;
            }
            else
                lpNewDIBBits[lWidth*j+i]=temp+100;
            if (temp<t)
                lpNewDIBBits[lWidth*j+i]=lpDIBBits[lWidth*j+i];
        }
    }
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
    // 释放内存
}

```

```

        LocalUnlock(hNewDIBBits);
        LocalFree(hNewDIBBits);
        return TRUE;
    }

```

灰度图像的拉普拉斯锐化算法实现的程序代码如下：

```

void CSharpenProcessDib::Laplacian( )
{
    int tempH;           //模板高度
    int tempW;           //模板宽度
    float tempC;         //模板系数
    int tempMY;          //模板中心元素 Y 坐标
    int tempMX;          //模板中心元素 X 坐标
    float Template[9];   //模板数组
    //设置拉普拉斯模板参数
    tempW=3;
    tempH=3;
    tempC=1.0;
    tempMY=1;
    tempMX=1;
    Template[0]=0.0;
    Template[1]=1.0;
    Template[2]=0.0;
    Template[3]=1.0;
    Template[4]=-4.0;
    Template[5]=1.0;
    Template[6]=0.0;
    Template[7]=1.0;
    Template[8]=0.0;
    //调用卷积函数
    Convolution(tempH,tempW,tempMX,tempMY,Template,tempC);
}

```

卷积运算函数实现的程序代码如下：

```

BOOL CSharpenProcessDib::Convolution(int tempH, int tempW, int tempMX, int tempMY,
float *fpTempArray, float fCoef)
{
    LPBYTE lpNewDIBBits;           // 指向缓存 DIB 图像的指针
    HLOCAL hNewDIBBits;
    long i,j,k,l;                  //循环变量
    float fResult;                 //暂存计算中间结果
    LPBYTE lpDIBBits=m_pDib->GetData(); //找到原图像的起始位置
    LONG lWidth=m_pDib->GetWidth(); //获得原图像的宽度
    LONG lHeight=m_pDib->GetHeight(); //获得原图像的高度
    // 暂时分配内存, 以保存新图像

```



```

hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);
if (hNewDIBBits == NULL)
{
    return FALSE;
}
lpNewDIBBits = (LPBYTE)LocalLock(hNewDIBBits);
// 初始化新分配的内存, 设定初始值为 0
memset(lpNewDIBBits, (BYTE)0, lWidth * lHeight);
// 逐个扫描图像中的像素点, 进行卷积运算
for(j=tempMY;j<lHeight-tempH+tempMY+1;j++)
{
    for(i=tempMX;i<lWidth-tempW+tempMX+1;i++)
    {
        // 计算像素值
        fResult=0;
        for(k=0;k<tempH;k++)
        for(l=0;l<tempW;l++)
            fResult=fResult+lpDIBBits[(j-tempMY+k)*lWidth+(i-tempMX+l)]
                                *fpTempArray[k*tempW+l];

        fResult*=fCoef; // 乘上系数
        fResult=(float)fabs(fResult); // 取绝对值
        // 判断是否超过 255
        if(fResult>255)
            // 若超过 255, 直接赋值为 255
            lpNewDIBBits[j*lWidth+i]=255;
        else
            // 未超过, 赋值为计算结果
            lpNewDIBBits[j*lWidth+i]=(BYTE)(fResult+0.5);
    }
}
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight); // 复制变换后的图像
// 释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);
return TRUE;
}

```

#### 4.4.5 彩色增强模块

彩色增强模块主要实现的是伪彩色增强部分, 具体程序代码如下:

```

void CColorEnhanceDib::Pseudo_Color_Enhance( BYTE * bpColorsTable )
{
    int i; // 循环变量
    DWORD wNumColors; // 颜色表中的颜色数目
}

```

```

        LPRGBQUAD m_lpRgbQuad;           // 指向图像颜色表的指针
        wNumColors = m_pDib->GetNumOfColor(); // 获取原图像颜色表中的颜色数目
        m_lpRgbQuad = m_pDib->GetRgbQuad(); // 获取原图像颜色表指针
// 判断颜色数目是否为 256 色
if (wNumColors == 256)
{
    // 读取伪彩色编码, 更新调色板
    for (i = 0; i < (int)wNumColors; i++)
    {
        // 更新调色板红色、绿色、蓝色分量
        (m_lpRgbQuad+i)->rgbBlue = bpColorsTable[i * 4];
        (m_lpRgbQuad+i)->rgbGreen = bpColorsTable[i * 4 + 1];
        (m_lpRgbQuad+i)->rgbRed = bpColorsTable[i * 4 + 2];
        // 更新调色板保留位
        (m_lpRgbQuad+i)->rgbReserved = 0;
    }
}
}

```

#### 4.4.6 模糊复原模块

实现运动图像复原的函数名为 DIBMotionRestore, 代码如下:

```

BOOL WINAPI DIBMotionRestore(CDib *pDib)
{
    BYTE * lpSrc;           // 指向源图像的指针
    LONG lWidth, lHeight;   // 图像的宽度和高度
    LONG lLineBytes;        // 图像每行的字节数
    // 得到图像的宽度和高度
    CSize SizeDim;
    SizeDim = pDib->GetDimensions();
    lWidth = SizeDim.cx;
    lHeight = SizeDim.cy;
    // 得到实际的 Dib 图像存储大小
    CSize SizeRealDim;
    SizeRealDim = pDib->GetDibSaveDim();
    lLineBytes = SizeRealDim.cx; // 计算图像每行的字节数
    LPBYTE lpDIBBits = pDib->m_lpImage; // 图像数据的指针
    // 循环变量
    long iColumn;
    long jRow;
    int i, n, m;
    int temp1, temp2, totalq, q1, q2, z; // 临时变量
    double p, q; // 临时变量
    int A = 80; // 常量 A 赋值
    int B = 10; // 常量 B 赋值
}

```



```

int nTotLen=10; //总的移动距离
// 图像宽度包含多少个 nTotLen
int K=lLineBytes/nTotLen;
int error[10];
int *nImageDegener; //用来存储源图像时域数据
nImageDegener = new int [lHeight*lLineBytes]; // 为时域数组分配空间
// 将像素存入数组中
for (jRow = 0; jRow < lHeight; jRow++)
{
    for(iColumn = 0; iColumn < lLineBytes; iColumn++)
    {
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow + iColumn;
        nImageDegener[lLineBytes*jRow + iColumn] = (*lpSrc);
    }
}
for (jRow = 0; jRow < lHeight; jRow++)
{
    // 计算 error[i]
    for(i = 0; i < 10; i++)
    {
        error[i] = 0;
        for(n = 0; n < K; n++)
            for(m = 0; m <= n; m++)
            {
                // 像素是否为一行的开始处
                if(i == 0 && m == 0)
                {
                    temp1=temp2=0;
                }
                // 进行差分运算
                else
                {
                    lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow
                        + m*10+i;
                    temp1=*lpSrc;
                    lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow
                        + m*10+i-1;
                    temp2 = *lpSrc;
                }
                error[i] = error[i] + temp1 - temp2;
            }
        error[i] = B * error[i] / K;
    }
    for(iColumn = 0; iColumn < lLineBytes; iColumn++)
    {
        // 计算 m, 以及 z
    }
}

```

```

m = iColumn / nTotLen;
z = iColumn - m*nTotLen;
// 初始化
totalq = 0;
q = 0;
for(n = 0; n <= m; n++)
{
    q1 = iColumn - nTotLen*n;
    if(q1 == 0)
        q = 0;
    // 进行差分运算
    else
    {
        q2 = q1 - 1;
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow + q1;
        temp1 = *lpSrc;
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow + q2;
        temp2 = *lpSrc;
        q = (temp1 - temp2) * B;
    }
    totalq = totalq + q;
}
p = error[z];
//得到 f(x,y) 的值
temp1 = totalq + A - p;
// 使得像素的取值符合取值范围
if(temp1 < 0)
    temp1 = 0;
if(temp1 > 255)
    temp1 = 255;
nImageDegener[lLineBytes*jRow + iColumn] = temp1;
}
}
//转换为图像
for (jRow = 0; jRow < lHeight ; jRow++)
{
    for(iColumn = 0; iColumn < lLineBytes ; iColumn++)
    {
        // 指向源图像倒数第 jRow 行, 第 iColumn 个像素的指针
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * jRow + iColumn;
        *lpSrc = nImageDegener[lLineBytes*jRow + iColumn]; //存储像素值
    }
}
delete nImageDegener; //释放存储空间
return true; // 返回
}

```

#### 4.4.7 滤镜效果模块

底片效果用 Negative 函数实现，其代码如下：

```
void Negative(BYTE* pixel, BYTE* tempPixel, int width, int height)
{
    int sum = width * height * 4;           //计算像素数组的长度
    memcpy(pixel, tempPixel, sum);
    for(int i = 0; i < sum; i += 4)         // 对像素值取反
    {
        tempPixel[i] = 255 - tempPixel[i];    //blue
        tempPixel[i+1] = 255 - tempPixel[i+1]; //green
        tempPixel[i+2] = 255 - tempPixel[i+2]; //red
    }
}
```

雕刻效果函数的实现代码如下。

```
void Emboss(BYTE* pixel, BYTE* tempPixel, int width, int height)
{
    int sum = width * height * 4; //计算像素数组的长度
    memcpy(tempPixel, pixel, sum);
    int r = 0, g = 0, b = 0;
    for(int i = 0; i < height-1; i++)
    {
        for (int j = 0; j < (width-1)*4; j+=4)
        {
            //处理像素值
            b = abs(tempPixel[i*width*4+j]-tempPixel[(i+1)*width*4+j+4]+128);
            g = abs(tempPixel[i*width*4+j+1]-tempPixel[(i+1)*width*4+j+5]+128);
            r = abs(tempPixel[i*width*4+j+2]-tempPixel[(i+1)*width*4+j+6]+128);
            if (r>255) //对于越界的像素值进行处理
                r=255;
            if (g>255)
                g=255;
            if (b>255)
                b=255;
            tempPixel[i*width*4 + j] = b;//blue
            tempPixel[i*width*4 + j + 1] = g;//green
            tempPixel[i*width*4 + j + 2] = r;//red
        }
    }
    for (int k = width * 4 * (height-1); k < width*4*height; k += 4)
    {
        tempPixel[k]=128;
        tempPixel[k+1]=128;
        tempPixel[k+2]=128;
    }
}
```

```

    }
    for (int l = (width-1) * 4; l < width*4*height; l += width*4)
    {
        tempPixel[l]=128;
        tempPixel[l+1]=128;
        tempPixel[l+2]=128;
    }
}

```

下面代码是黑白效果函数的实现。

```

void ColorToBW(BYTE* pixel, BYTE* tempPixel, int width, int height)
{
    int sum = width * height * 4; //计算像素数组的长度
    memcpy(tempPixel, pixel, sum);

    for(int i = 0; i < sum; i += 4)
    {
        //平均值法
        tempPixel[i] = (tempPixel[i] + tempPixel[i+1] + tempPixel[i+2]) / 3; //blue
        tempPixel[i + 1] = tempPixel[i]; //green
        tempPixel[i + 2] = tempPixel[i]; //red
        //最大值法
        /*
        tempPixel[i] = tempPixel[i] > tempPixel[i+1]?
            tempPixel[i] : tempPixel[i+1];
        tempPixel[i] = tempPixel[i] > tempPixel[i+2]?
            tempPixel[i] : tempPixel[i+2];
        tempPixel[i+1] = tempPixel[i];
        tempPixel[i+2] = tempPixel[i];
        */
        //加权平均值法
        /*
        tempPixel[i] = ( ((int) (tempPixel[i]*0.5)) +
            ((int) (tempPixel[i+1]*0.3)) +
            ((int) (tempPixel[i+2]*0.2)) );
        tempPixel[i + 1] = tempPixel[i]; //green
        tempPixel[i + 2] = tempPixel[i];
        */
    }
}

```

水平方式的雾化效果函数 HorFog 的代码如下：

```

//-----
//作用： 实现水平方式的雾化效果
//参数：
// pixel      原始图像的像素数组

```

```

// tempPixel    输出图像的像素数组
// width        原始图像宽度
// height       原始图像高度
//-----

void HorFog(BYTE* pixel, BYTE* tempPixel, int width, int height, int f)
{
    int k;
    int n;
    for(int i = 0; i < height; i++)
    {
        for (int j = 0; j < width*4; j += 4)
        {
            k = abs(rand() % f);
            n = j + k*4;
            if (n > (width-1) * 4)
                n = (width-1)*4;
            tempPixel[i*width*4 + j] = pixel[i*width*4 + n];
            tempPixel[i*width*4 + j + 1] = pixel[i*width*4 + n + 1];
            tempPixel[i*width*4 + j + 2] = pixel[i*width*4 + n + 2];
            tempPixel[i*width*4 + j + 3] = pixel[i*width*4 + n + 3];
        }
    }
}

```

垂直方式的雾化效果函数 VerFog 的程序代码如下:

```

//-----
//作用: 实现垂直方式的雾化效果
//参数:
//pixel    原始图像的像素数组
//tempPixel 输出图像的像素数组
//width    原始图像宽度
// height  原始图像高度
//-----

void VerFog(BYTE* pixel, BYTE* tempPixel, int width, int height, int f)
{
    int k;
    int m;
    for(int i = 0; i < height; i++)
    {
        for (int j = 0; j < width*4; j += 4)
        {
            k = abs(rand() % f);
            m = i + k;
            if (m > height-1)
                m = height-1;

```

```

tempPixel[i*width*4 + j]    = pixel[m*width*4 + j];
tempPixel[i*width*4 + j + 1] = pixel[m*width*4 + j + 1];
tempPixel[i*width*4 + j + 2] = pixel[m*width*4 + j + 2];
tempPixel[i*width*4 + j + 3] = pixel[m*width*4 + j + 3];
    }
}
}

```

复合方式的雾化函数 ComFog 的程序代码如下：

```

//-----
//作用： 实现复合方式的雾化效果
//参数：
//pixel      原始图像的像素数组
//tempPixel  输出图像的像素数组
//width      原始图像宽度
//height     原始图像高度
//-----
void ComFog(BYTE* pixel, BYTE* tempPixel, int width, int height, int f)
{
    int k;
    int m, n;
    for(int i = 0; i < height; i++)
    {
        for (int j = 0; j < width*4; j += 4)
        {
            k = abs(rand() % f);
            m = i + k;
            n = j + k * 4;
            if (m > height-1) // 对超出图像区域的点进行相应处理
                m = height-1;
            if (n > (width-1) * 4)
                n = (width-1) * 4;
            tempPixel[i*width*4 + j]    = pixel[m*width*4 + n]; // 更新像素数组
            tempPixel[i*width*4 + j + 1] = pixel[m*width*4 + n + 1];
            tempPixel[i*width*4 + j + 2] = pixel[m*width*4 + n + 2];
            tempPixel[i*width*4 + j + 3] = pixel[m*width*4 + n + 3];
        }
    }
}

```

下面是素描模板函数 TempltExcute( )的实现代码。

```

//-----
// 功能：使用模板对彩色图邻域进行运算
// 参数：
// imageBuf 为目标图像
// w、h 为图像大小

```



## 数字图像处理典型案例详解

```

// templt 为模板
// tw 为邻域大小
// x,y 为要取得像素的坐标
// cn 为颜色分量编号 0 为蓝色 1 为绿色 2 为红色
//-----
int TempltExcuteCl(BYTE** imageBuf0, int w, int h, int* templt, int tw, int x, int y,
                  int cn)
{
    int i,j;                //循环变量
    int m=0;                //用来存放加权和
    int px,py;
    for(i=0; i<tw; i++)    //依次对邻域中每个像素进行运算
    {
        for(j=0; j<tw; j++)
        {
            py=y-tw/2+i;    //计算对应模板上位置的像素在原图像中的位置
            px=x-tw/2+j;
            m+=imageBuf0[py][px*4+cn] * templt[i*tw+j]; //加权求和
        }
    }
    return m; //返回结果
}

```

素描函数的实现代码如下：

```

//-----
//作用： 实现图像的素描效果
//参数：
// pixel          原始图像的像素数组
// tempPixel      输出图像的像素数组
// w              原始图像宽度
// h              原始图像高度
//-----
void LaplacianB(BYTE* pixel, BYTE* tempPixel, int w, int h)
{
    BYTE* tempImage; //定义临时图像存储空间
    tempImage = (BYTE*)malloc(sizeof(BYTE)*w*h*4);
    BYTE** imageBuf0 = CreatImage(pixel, w, h); //将图像转化为矩阵形式
    BYTE** imageBuf1 = CreatImage(tempPixel, w, h);
    BYTE** tempImageBuf = CreatImage(tempImage, w, h);
    double scale = 2;
    int templt[9]={ 1, 1, 1, 1,-8, 1, 1, 1, 1 }; //拉普拉斯正相模板
    int templtTest1[9]={ 1, 1,-1, 1, 0,-1, 1,-1,-1 }; //噪声检测模板
    int templtTest2[9]={ 1, 1, 1,-1, 0, 1,-1,-1,-1 };
    int templtAve[9]={ 1, 1, 1, 1, 4, 1, 1, 1, 1 }; //模糊处理模板
    int x,y;
    int a,b,b1,b2;
}

```

```

for(y = 1; y < h - 1; y++) //依次对原图像的每个像素进行处理
    for(x = 1; x < w - 1; x++)
    {
        a=TempltExcute(imageBuf0, w, h, templt, 3, x, y); //拉普拉斯卷积运算
        b1=abs(TempltExcute(imageBuf0, w, h, templtTest1, 3, x, y)); //噪声检测
        b2=abs(TempltExcute(imageBuf0, w, h, templtTest2, 3, x, y));
        b=b1>b2?b1:b2;
        if(b<25) a=0;
        else
        {
            a = (int)(a * scale);
            if(a > 255) a = 255; //过限处理
            else if(a < 32) a=0;
        }
        a=255-a; //反色处理
        SetPixel2(tempImageBuf, x, y, a);
    }
for(y = 1; y < h - 1; y++) //模糊处理
    for(x = 1; x < w - 1; x++)
    {
        a=TempltExcute(tempImageBuf, w, h, templtAve, 3, x, y) / 12;
        SetPixel2(imageBuf1, x, y, a);
    }
free(tempImage);
free(imageBuf0);
free(imageBuf1);
free(tempImageBuf);
}

```

## 4.5 经验分享

1) 图像增强要根据需求选择合适的方法。图像增强的处理是面向问题的，不以保真为原则，根据图像的一些性质提出改善算子，其目的是改善图像的质量，让观察者得到直观、清晰、适合于分析的依据。所以要根据特定的目标、特定的应用来选择特定的方法。

2) 编写程序时的注意事项。在编写代码时要注意除法运算的两个操作数中必须至少有一个是 float 或 double 类型，这样才能保证除法运算的准确性。本章中有一些算法的程序中就涉及除法运算，如直方图统计、邻域均值平滑等。如果在做除法时不做类型转换，将导致计算结果的不准确，甚至出现运行结果图像漆黑一片，所以在编写程序时一定要注意这个问题。

## 第 5 章 CT 图像重建系统

相传战神之子 Romulus 在一夜之间建立了罗马城，留下了“Rome was not built in a day”（罗马城不是一个白天建起来的）美丽传说，这句话流传至今却演变成了“罗马城不是一天建起来的”。而图像三维重建技术则可以让罗马城真的在一天就重建起来，只要有足够的数据。三维重建就是利用图形图像技术将数据重构成三维模型或场景的过程，在数字考古、医疗诊断、辅助设计等领域具有广泛的应用。本章以一个 CT 图像重建系统为例，来解读三维重建的基本方法。

本章要点：

- 三维可视化技术
- 图像重建技术
- CT 图像重建系统功能描述
- CT 图像重建系统的总体结构和主要流程
- CT 图像重建系统的编程实现

### 5.1 核心技术原理

X 射线计算机断层成像（X-ray Computed Tomography，简称 CT）技术是 20 世纪 80 年代发展起来的一种融合了射线光电子学、信息科学、微电子学和计算机科学等领域知识的高新技术。CT 是目前最先进的无损检测技术之一，广泛应用于航空、航天、工业、诊断医学和生物等多个领域的检测评估。

X 射线断层成像技术的物理原理是：X 射线穿过物体时会发生强度衰减，衰减的程度和物体的密度、原子序数、入射射线的能量和初始强度有关。利用这个原理，让 X 射线从不同的角度穿过被检测物体并被探测器接收，再利用各种不同的算法重建出被检测物体各处的射线衰减系数并得到其分布图像，以此反映被测物体的密度分布，这就是 CT 技术的基本思想。

图像重建系统模型如图 5-1 所示，其中探测结果可以是二维数据信息，也可以是三维数据信息。根据不同的重建算法，得到相应的不同衰减系数，再通过衰减系数将图像重建进而得到分布图像。

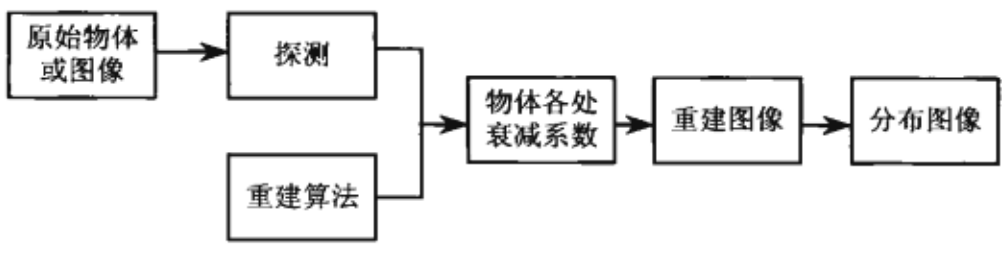


图 5-1 图像重建系统模型图

5.1.1 三维可视化技术

科学计算可视化是 1987 年由 B.H. McCormick 等人根据美国国家科学基金会召开的“科学计算可视化研讨会”的内容撰写的一份报告中正式提出的。随后，美国、西欧和日本各著名大学、研究所、超级计算机中心以及各大公司纷纷进行科学计算可视化理论和方法的研究，科学计算可视化迅速成为计算机科学中一个热门的研究领域。

可视化是运用计算机图形学和图像处理技术，将科学计算过程中及计算结果的数据转换为图形及图像在屏幕上显示出来并进行交互处理的理论、方法和技术。近些年来随着计算机硬件水平的提高和可视化理论方法的不断完善，可视化在许多领域都得到了广泛的应用，如有限元分析、CT 及核磁共振 MRI 数据的可视化等。

科学计算可视化技术的核心是三维空间数据的可视化。三维数据可视化的基本流程如图 5-2 所示。

其中，第一步是开始载入，第二步是数据生成。即可由计算机数值模拟或测量仪器产生数据。计算机数值模拟的结果形成数据文件，文件格式由科学计算工作者来定义，因而它是已知的，可以比较方便地输入计算机。但是，有些测量仪器输出数据的文件格式却是不公开的（例如，大多数 CT 或 MRI 设备就是如此），必须得到仪器制造厂家的配合才能弄清楚数据文件格式，将其输入计算机。

流程中的第三步是数据的精炼与处理。因为应用对象不同，这一步的功能也会各不相同。对于数据量过大的原始数据，需要加以精炼和选择，以适当减少数据量。例如旋转 CT 扫描的层间距很小，仅为 1mm。图像的分辨率也很高，通常为  $512 \times 512$ ，而且原始数据的灰度等级为 4096，因此，一组  $512 \times 512 \times 100$  的 CT 扫描图像其数据量为 53 兆字节。这时需要对原始数据加以精炼和选择，既要减少数据量，又要最大限度地减少有用信息的丢失。

相反地当数据分布过分稀疏而有可能影响可视化的效果时，需要进行有效的插值处理。这一步中

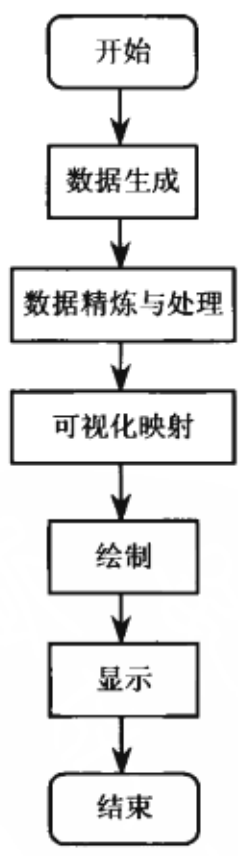


图 5-2 三维数据可视化流程图

最常见的处理方法是消除噪声、参数域变换以及法向量计算等。因为原始数据中一般不会包括数据点所在处的法向，而这又是在后续步骤中需要用到的，因而需要预先计算出来。

流程中的第四步是可视化映射。这里“映射”的含义包括可视化方案的设计，既需要决定在最后的图像中应该看到什么，又需要决定如何将其表现出来。也就是说，如何用形状、光度、颜色以及其他属性表示出原始数据中人们感兴趣的性质和特点。这时，往往有多种实现方案，可供科学计算工作者自主选择。

流程的第五步将第三步产生的几何图素和属性转换为可供显示的图像，所用的方法是计算机图形学中的基本技术，包括视觉变换、光照计算、隐蔽面消除以及扫描变换等。在图形工作站上，可以借助已有的图形软件包及图形硬件完成以上的功能。

流程的第六步是图像的变换和显示。包括图像的几何变换、图像压缩、颜色量化、图像格式转换以及图像的动态输出等，之后进入结束状态。

通常从三维图像数据重建其组织结构的方法大致可以分为体积方法和表面方法两大类。体积方法根据特定的准则和处理方法将体积数据分割为对应感兴趣结构的体积区域，并形成标记体积图像。表面方法则与二维图像处理中的边界提取方法有些类似，从体积图像数据中提取出特定的结构表面。

### 1. 三维图像的面绘制

表面表示是表示三维物体形状最基本的方法，它可以提供三维物体的表面信息。它的基本思想是先对体数据中待显示的物体表面进行分割，然后通过几何单元内插形成物体表面，最后通过光照、明暗模型进行渲染和消隐后得到显示图像。在计算机图形学领域，面绘制算法目前已经发展到较为成熟的阶段，其具体形式有两种：边界轮廓线表示法和表面曲面表示法。最初的面绘制方法是采用基于轮廓线的方法，即首先通过分割提取二维断层图像轮廓线，然后把各层对应的轮廓线拼接在一起表示感兴趣物体的表面边界，这样的算法相对简单但显示结果比较粗糙，不够直观。基于表面曲面的表示方法是由轮廓重建物体的表面，用三角形或多边形的小平面（或曲面）在相邻的边界轮廓线间通过特定的算法填充形成物体的表面，MC（Marching Cubes）方法便是表面提取方法中公认的最有效的一种，下面对其基本原理做简要阐述。

在 MC 算法中，假定原始数据是离散的三维空间规则数据场，用于医疗诊断的 CT 及 MRI 等产生的图像属于这一类型。MC 算法的基本思想是逐个处理数据场中的立方体（体素），分类出与等值面相交的立方体，采用插值计算出等值面与立方体边的交点。根据立方体每一顶点与等值面的相对位置，将等值面与立方体边的交点按一定方式连接生成等值面，作为等值面在该立方体内的一个逼近表示。因而，MC 算法中每一单元内等值面抽取的两个主要计算是体素中由三角片逼近的等值面计算和三角片各顶点法向量计算。

#### （1）体素中等值面剖分方式的确定

MC 算法的基本假设是沿着立方体的边数据场呈连续线性变化，也就是说，如果一条边的两

个顶点分别大于、小于等值面的值，则在该边上必有也仅有一点是这条边与等值面的交点。确定立方体体素中等值面的分布是该算法的基础。首先对立方体的八个顶点进行分类，以判定其顶点是位于等值面之外，还是位于等值面之内。然后，再根据八个顶点的状态，确定等值面的剖分模式。

顶点分类规则为：

- 1) 如立方体顶点的数据值>等值面的值，则定义该顶点位于等值面之外，记为“0”。
- 2) 如立方体顶点的数据值<等值面的值，则定义该顶点位于等值面之内，记为“1”。

由于每一体素共有 8 个顶点，每个顶点共有 2 个状态，因此共有 256 种组合状态。根据互补对称性，即体素的顶点标记置反（0 变为 1，1 变为 0），不影响该体素内三角面片的拓扑结构，这样 256 种构型可以简化成 128 种。再根据旋转对称性，可将这 128 种构型进一步简化成 15 种。图 5-5 给出了这 15 种基本构型的三角剖分，其中黑点表示标记为 1 的角点。对于 8 个角点的标记都为 1 或者都为 0 的体素，它属于“0”号构型，没有等值面穿过该体素。当只有一个角点标记为 1 时，即“1”号构型，我们用一个三角面片代表体素内的等值面片，它将该角点与其他 7 个角点分成两部分。对于其余几种构型，将产生多个三角面片。在构型的处理中，首先建立一个“构型-三角剖分”查找表，它包含 256 个索引项。每个索引项包含索引、旋转和三角部分模式（构型）。索引项结构如图 5-3 所示。

索引	旋转	三角剖分模式（构型）
----	----	------------

图 5-3 索引项结构图

其中，索引（Index）是体素 8 个角点标记的有序二进制编码，二进制编码顺序如图 5-4 所示。

V <sub>0</sub>	V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>	V <sub>5</sub>	V <sub>6</sub>	V <sub>7</sub>
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------

图 5-4 二进制编码顺序图

具体的方法是：对于每个体素，首先根据它的索引在“构型-三角剖分”查找表中确定其三角剖分形式，然后再根据相应索引项中的旋转参数确定最终的三角剖分。

### （2）等值面与体素边界的交点

在确定立方体内三角剖分模式后，就要计算三角片顶点位置。当三维离散数据场的密度较高时，即当体素很小时，可以假定函数值沿体素边界呈线性变化，这就是 MC 算法的基本假设。根据这一基本假设，可以直接用线性插值计算等值面与体素边的交点。

对于某棱边，如果它的两个端点  $v_1$ 、 $v_2$  标记不同，那么等值面一定与此棱边相交。

- 1) 体素棱边与 X 轴平行时，设该边的两端点为  $v_1(i, j, k)$ ， $v_2(i+1, j, k)$ ，则交点为  $v(i, j, k)$ 。

$$x = i + \frac{[c - f(v_1)]}{[f(v_2) - f(v_1)]} \tag{5-1}$$



2) 体素棱边与  $Y$  轴平行时, 设该边的两端点为  $v_1(i, j, k)$ ,  $v_2(i+1, j, k)$ , 则交点为  $v(i, j, k)$ 。

$$y = j + \frac{[c - f(v_1)]}{[f(v_2) - f(v_1)]} \quad (5-2)$$

3) 体素棱边与  $Z$  轴平行时, 设该边的两端点为  $v_1(i, j, k)$ ,  $v_2(i, j+1, k)$ , 则交点为  $v(i, j, k)$ 。

$$z = k + \frac{[c - f(v_1)]}{[f(v_2) - f(v_1)]} \quad (5-3)$$

求出了等值面与体素棱边的交点以后, 根据索引表确定的三角剖分, 即可将这些交点连接成三角片, 得到该体素内的等值面片。等值面体一共包含 15 种构型, 如图 5-5 所示。

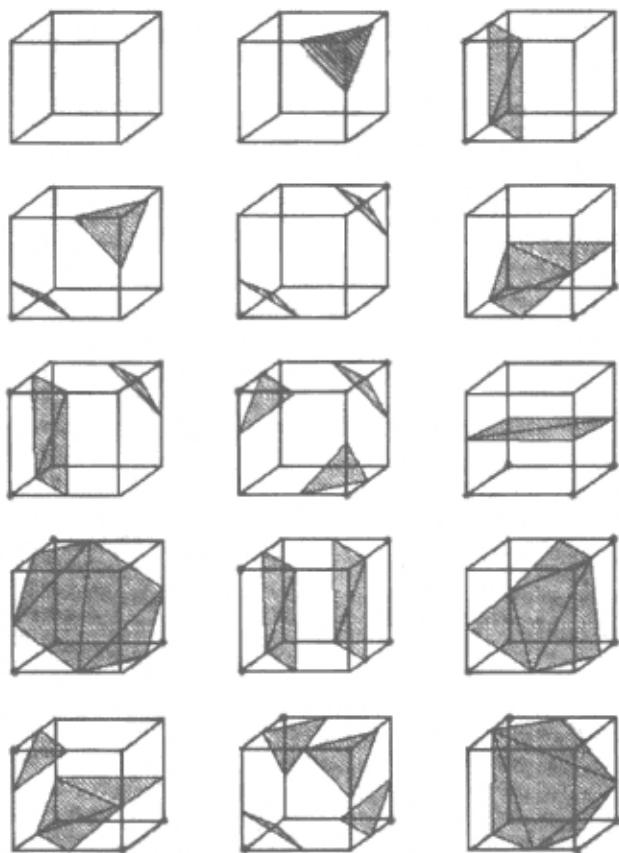


图 5-5 立方体中的等值面体的 15 种构型

### (3) 等值面的法向计算

为了利用图形硬件显示等值面图像, 必须给出形成等值面的各三角面片的法向, 选择适当的光照模型进行光照计算, 生成真实感图形。

对于等值面上的每一点, 其沿面的切线方向的梯度分量应该是零。因此, 该点的梯度矢量的方向也就代表了等值面在该点的法向, 而且等值面往往是两种具有不同密度的物质的分界面, 因而其梯度矢量不为零值, 即

$$g(x, y, z) = \nabla f(x, y, z) \quad (5-4)$$

直接计算三角面片的法向是费时的, 而且, 为了消除各三角面片之间明暗度的不连续变化,

只要给出三角面片各顶点处的法向并采用哥罗德 (Gouraud) 模型绘制各三角面片就行了。

本书采用中心差分计算出体素各角点处的梯度, 然后再一次通过体素棱边两个端点处梯度的线性插值求出三角面片各顶点的梯度, 也就是各顶点处的法向, 从而实现等值面的绘制。体素角点的中心差分公式如下:

$$\begin{cases} g_x = \frac{f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)}{2U_x} \\ g_y = \frac{f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)}{2U_y} \\ g_z = \frac{f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})}{2U_z} \end{cases} \quad (5-5)$$

其中,  $U_x$ 、 $U_y$  和  $U_z$  分别是体素的边长。

#### (4) MC 算法抽取等值面的算法流程

MC 算法求等值面的步骤如下:

- 1) 将三维离散规则数据场分层读入。
- 2) 扫描两层数据, 逐个构造体素, 每个体素中的 8 个角点取自相邻的两层。
- 3) 将体素每个角点的函数值与给定的等值面值  $c$  做比较, 根据比较结果, 构造该体素的索引表。
- 4) 根据索引表得出将与等值面有交点的边界体素。
- 5) 通过线性插值方法计算出体素棱边与等值面的交点。
- 6) 利用中心差分方法, 求出体素各角点处的法向量, 再通过线性插值方法, 求出三角面片各顶点处的法向。
- 7) 根据各三角面片各顶点的坐标及法向量绘制等值面图像。

## 2. 三维图像的体绘制

体绘制技术的中心思想是为每一个体素指定一个不透明度 (Opacity), 由光线穿过整个数据场, 并考虑每一个体素对光线的透射、发射和反射作用, 这里体素就是将三维图像中的每一像素看成是空间中的一个六面体单元。光线的透射取决于体素的不透明度; 光线的发射取决于体素的物质度 (Objectness), 物质度越大, 其发射光越强; 光线的反射则取决于体素所在的面与入射光的夹角关系。体绘制的步骤原则上可分为投射、消隐、渲染和合成 4 个步骤。体绘制算法按处理数据域的不同可分为空间域方法和变换域方法。前者是直接对原始的体数据进行处理显示; 后者是将体数据变换到变换域, 然后再进行处理显示。

基于空间域的经典方法主要有光线跟踪法 (Ray Casting)、抛雪球法 (Splatting) 和错切-形变法 (Shear-warp) 等。基于变换域的方法主要有频域体绘制法 (Frequency Domain-Volume Rendering) 和基于小波的体绘制法 (Wavelet-based Volume Rendering) 等。下面对这些方法做简要的介绍。

### (1) 光线跟踪法 (Ray casting)

这种方法是 Levoy 于 1988 年提出的基于图像空间体绘制的经典算法, 是一种以图像空间为序的体绘制方法。将每个体素都看成能够透射、发射和反射光线的粒子, 从图像空间的每一体素出发, 根据设定的方向反射一条光线, 在其穿过各个切片组成的空间域的过程中, 等间距地进行二次采样, 由每个二次采样点的 8 个邻域体素用三次线性插值法得到采样点的颜色和阻光度值, 然后依据体素的介质特性得到它们的颜色 (灰度图像为亮度) 和不透明值, 从而在屏幕上得到三维数据图像。

光线跟踪法的缺点是需要遍历每个体素, 计算量比较大, 而且当观察方向发生变化时, 数据场中的采样点之间的前后关系也必然会变化, 这样就要重新进行采样。为解决这一缺点, 一般可利用光线提前终止和空间相关性等方法来提高算法的效率。

### (2) 抛雪球法 (Splatting)

抛雪球法也称足迹法, 首先是由 Westover 提出的, 它模仿了雪球被抛到墙壁上所留下的一个扩散状痕迹的现象, 因而得名“抛雪球法”。其原理是将体数据表示为一个由交迭的基本函数 (这里基本函数通常选择幅值由体素值表示的高斯函数核) 构成的矩阵, 然后根据一个预先计算的通用足迹查询表, 把这些基本函数投射到像平面以生成图像。其实质也可以看成为将体数据与函数核作卷积, 再沿视线的反方向投射积累到像平面的过程。抛雪球法的最大优点是只有与图像有关的体素才会被映射到像平面, 从而大大地减少了需要处理和存储的数据量。

对于光线跟踪法来说, 当观察方向发生变化后, 就需要重新采样并插值计算, 而对于抛雪球法, 若视线方向发生变化, 就需要重新计算重构核空间卷积域的椭圆投影, 投影中的每个像素都需要进行旋转和比例变换, 以便查找通用足迹查询表, 因此针对上述问题提出了错切-形变法 (Shear-warp) 算法。

### (3) 错切-形变法 (Shear-warp) 算法

这一算法最早是由 Cameron 和 Undrill 提出的, 它采用一种关于体素和图像的编码方案, 在遍历体素和图像的同时可以略去不透明的图像区域和透明体素, 被认为是一种速度最快的体绘制算法。其基本思想包括 3 个步骤: 首先将体数据变换到错切后的物体空间, 并对每一层数据采样。其次, 按从前至后的顺序将体数据投射到二维中间图像平面。最后通过变形变换, 将中间图像投影到像空间产生最终的图像。错切-形变法的缺点是在视域的分辨率高于体素分辨率的情况下, 所绘制的图像品质会明显下降。

### (4) 基于硬件的三维纹理映射 (3DTexture-MappingHardware) 方法

这种绘制方法首先由 Cabral 应用于无明暗处理的体绘制, 三维纹理映射方法是基于硬件来提高体绘制的速度, 需要专用的图像硬件。首先将数据值进行分类, 并按照给定的转换函数将每个数据点转换成相应的颜色值及不透明值, 形成三维纹理图; 其次, 在确定了观察方向之后, 给出被绘制数据场中的采样点与纹理空间的映射关系; 最后在纹理空间中进行采样, 然后再进行图像

合成,形成最终的三维图像。这类方法在物体表面细节十分复杂和精细时显得尤为重要。

体绘制技术由于直接研究光线通过体数据场时与体素的相互关系,无须构造中间面。体素中的许多细节信息得以保留,结果的保真性大为提高。因而从绘制结果来讲,体绘制的图像质量通常要优于面绘制。但是体绘制法对硬件的要求很高,运行速度较慢。

VTK 为使用者提供了两种体绘制函数分别是:最大密度投影函数(vtkVolumeRayCastMIPFunction)和合成体绘制函数(vtkVolumeRayCastCompositeFunction)。最大密度投影(Maximum Intensity Projection, MIP)是一种应用广泛的 CT 及 MR 图像后处理技术。

MIP 运用透视法获得二维图像,即通过计算沿着被扫描物体每条射线上所遇到的最大密度像素而产生的。当光线束通过一段组织的原始图像时,图像中密度最大的像素被保留,并被投影到一个二维平面上,从而形成 MIP 重建图像。MIP 能反映相应像素的 X 线衰减系数,较小的密度变化也能在 MIP 图像上显示,能很好地显示血管的狭窄、扩展、充盈缺损及区分血管壁上的钙化与血管腔内的对比剂。

MIP 的算法实现如下(在此,主要是针对轴状位图,其他方向的 MIP 算法大体相同):

建立一个直角坐标系,  $x_{pixel}(0-newWidth)$  为新图像中像素的坐标,  $[x_0-Width, y_0-Height]$  则为原始图像中像素的横纵坐标。

在  $\theta \in (0^\circ, 90^\circ)$  时,可以得到如下方程:

$$y = \tan(\theta + 90^\circ) \left( x - \frac{x_{pixel}}{\cos \theta} \right) \quad (5-6)$$

从式(5-6)中推出下式:

$$x = \frac{y - \frac{x_{pixel}}{\sin \theta}}{\tan(\theta + 90^\circ)} + 0.5 \quad (5-7)$$

而当在  $\theta \in (90^\circ, 180^\circ)$  时,方程需要稍微修改如下:

$$y = \tan(\theta - 90^\circ) \left( x - Width - \frac{x_{pixel}}{\sin(\theta - 90^\circ)} \right) \quad (5-8)$$

从式(5-8)中推出下式:

$$x = \frac{y}{\tan(\theta - 90^\circ)} + Width - \frac{x_{pixel}}{\sin(\theta - 90^\circ)} \quad (5-9)$$

将  $y$  从 0 逐渐递增到  $Height$ ,便可以遍历整个原始图像的像素值,得到投影光线与图像所有交点的像素值,因为求得的  $x$  不一定为整数,本书采用最简单的插值方式,取最邻近值点作为  $x$  的值。

当  $\theta$  从  $0^\circ$  至  $180^\circ$  变化时,在屏幕上看到的图像就会忽大忽小且位置不断变化。经过分析可以看出,新图像宽度的最大值为:  $\sqrt{Width^2 + Height^2}$ , 可以将产生的新图像宽度全部设置为最大宽度,则最大宽度与实际宽度之间就会有个偏移量,值为  $\sqrt{Width^2 + Height^2} - Width \cos \theta - Height \sin \theta / 2$ ,

因此在新图像赋值时,先将新图像数据指针加上偏移量再赋值,这样就可以保证产生的新图像的大小一样,且中心位置固定,看起来有连续的效果。

### 5.1.2 图像重建技术

传统的图像重建算法主要有傅里叶反投影重建、卷积反投影重建、代数重建和超分辨率重建等,尽管有些算法现在已经不太常用,但是了解其基本思想对于学习和研究图像重建技术也不无裨益。

#### 1. 傅里叶反投影重建

傅里叶反投影重建方法可以说是最简单的一种变换重建方法。该重建方法最早于 1974 年由 Shepp 和 Logan 提出,该方法是建立在“对于任何一个三维(二维)物体,它的二维(三维)投影的傅里叶变换恰好是该物体的傅里叶变换的主体部分”这一理论基础上的,关于这一理论的推导和证明过程将在稍后给出。根据此结论,可以先对投影进行旋转和傅里叶变换以构造整个傅里叶变换的平面,然后再对其进行傅里叶反变换即可得到重建后的目标。

傅里叶反投影重建方法由于是一种变换重建方法,因此首先应该对其在连续域内进行解析处理,最后再进行离散化处理,以便可以借助计算机进行处理。具体而言,投影重建图像应首先建立好以连续实函数形式给出的数学模型,然后利用反变换公式求解未知量,并适当调节反变换公式以适应在离散、有噪声干扰等条件下的应用需求。

傅里叶反投影重建的理论基础是傅里叶变换投影定理,该定理描述为:图像函数  $f(x, y)$  在  $X$  轴成  $\theta$  角的直线上的投影  $g(s, \theta)$  的傅里叶变换是  $f(x, y)$  的傅里叶变换在朝向角  $\theta$  上的一个截面。假定  $G(r, \theta)$  是  $g(s, \theta)$  关于变量  $s$  的一维傅里叶变换,存在如下关系:

$$G(r, \theta) = \int_{-\infty}^{+\infty} g(s, \theta) e^{-j2\pi rs} ds \quad (5-10)$$

对于  $f(x, y)$  的二维傅里叶变换:

$$F(u, v) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-j2\pi(ux+vy)} dx dy \quad (5-11)$$

则存在关系:

$$G(r, \theta) = F(r \cos \theta, r \sin \theta) \quad (5-12)$$

对于式 (5-12) 可证明如下:

图像  $f(x, y)$  在  $X$  轴上的投影,可沿用  $Y$  轴的积分来表示:

$$g_y(x) = \int_{-\infty}^{+\infty} f(x, y) dy \quad (5-13)$$

此投影的一维傅里叶变换为:

$$\begin{aligned} G_y(u) &= \int_{-\infty}^{+\infty} g_y(x) e^{-j2\pi ux} dx \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-j2\pi ux} dx dy \\ &= F(u, 0) \end{aligned} \quad (5-14)$$

如果图像函数并没有投影到  $X$  轴而是投影到一条旋转了  $\theta$  角的直线上, 那么可以首先定义旋转坐标如下:

$$\begin{cases} s = x \cos \theta + y \sin \theta \\ t = -x \sin \theta + y \cos \theta \end{cases} \quad (5-15)$$

将函数投影直线选为  $X$  轴, 根据投影关系, 投影点将沿直线  $s_1 = x \cos \theta + y \sin \theta$  对距离  $t$  轴  $s_1$  处的平行线进行函数积分:

$$g(s_1, \theta) = \int_{s_1} f(x, y) ds_1 \quad (5-16)$$

对投影  $g(s_1, \theta)$  进行傅里叶变换, 并将式 (5-16) 代入, 可得:

$$\begin{aligned} G(r, \theta) &= \int_{-\infty}^{+\infty} g(s_1, \theta) e^{-j2\pi r s_1} ds_1 \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-j2\pi r (x \cos \theta + y \sin \theta)} dx dy \\ &= F(r \cos \theta, r \sin \theta) \end{aligned} \quad (5-17)$$

从而得证。

傅里叶变换投影定理只是进行傅里叶反投影重建的理论基础, 具体的重建工作还需要根据傅里叶反投影重建公式来进行。下面从式 (5-17) 出发继续对其进行推导, 首先进行变量替换:

$$\begin{cases} u = r \cos \theta \\ v = r \sin \theta \end{cases} \quad (5-18)$$

同时可以得到用  $u$  和  $v$  表示的  $r$  和  $\theta$ :

$$\begin{cases} r = \sqrt{u^2 + v^2} \\ \theta = \arctan\left(\frac{v}{u}\right) \end{cases} \quad (5-19)$$

代入到式 (5-17) 可得:

$$\begin{aligned} F(u, v) &= G(r, \theta) \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(x, y) e^{-j2\pi(ux+vy)} dx dy \end{aligned} \quad (5-20)$$

对上式进行傅里叶反变换, 并将式 (5-19) 代入, 可求出原图像函数  $f(x, y)$ :

$$\begin{aligned} f(x, y) &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} F(u, v) e^{j2\pi(ux+vy)} du dv \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} G(r, \theta) e^{j2\pi(ux+vy)} du dv \\ &= \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} G(\sqrt{u^2 + v^2}, \arctan(v/u)) du dv \end{aligned} \quad (5-21)$$

由于在公式 (5-17) 中  $G(r, \theta)$  是  $f(x, y)$  的投影  $g(s_1, \theta)$  的傅里叶变换, 而投影函数  $g(s_1, \theta)$  可以从实验中获取, 因此公式 (5-17) 可以作为傅里叶反投影的重建公式。



## 2. 卷积反投影重建

上节介绍的傅里叶反投影重建方法是建立在傅里叶投影定理基础之上的, 本节介绍的卷积反投影重建技术也以傅里叶投影定理为基础。所不同的是, 傅里叶反投影重建方法在进行二维傅里叶变换时是用笛卡儿平面直角坐标系表示的, 而卷积反投影重建技术则是建立在极坐标基础之上的。在推导卷积反投影重建公式之前, 首先, 对笛卡儿坐标系和极坐标系之间的相互关系做一简要介绍。

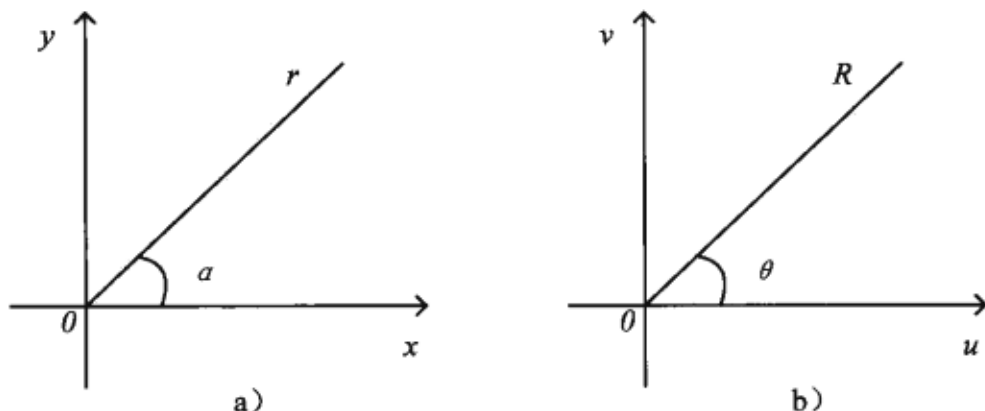


图 5-6 笛卡儿坐标系和极坐标系在空间域和变换域中的对应关系

图 5-6a 和 b 分别给出了笛卡儿坐标系和极坐标系在空间域和变换域中的对应关系。其坐标形式为:

$$\begin{cases} x = r \cos \alpha \\ y = r \sin \alpha \end{cases} \quad (5-22)$$

$$\begin{cases} u = R \cos \theta \\ v = R \sin \theta \end{cases} \quad (5-23)$$

将式 (5-22) 和 (5-23) 代入式 (5-21) 中, 可得到傅里叶投影变换定理的极坐标形式为:

$$f(r, \alpha) = \int_0^{2\pi} \int_{-\infty}^{+\infty} G(R, \theta) \cdot R \cdot e^{j2\pi Rr \cos(\alpha - \theta)} dR d\theta \quad (5-24)$$

$G(R, \theta)$  可由投影函数的  $g(s, \theta)$  一维傅里叶变换得到, 因此, 上式也可以作为投影重建图像的公式。在实际应用此公式时, 也应像傅里叶反投影重建方法一样, 将积分限制在一个有限的范围内进行。根据采样定理, 在有限带宽  $-\frac{1}{2\Delta s} < R < \frac{1}{2\Delta s}$  的情况下对  $G(R, \theta)$  进行估计, 此时可得到原图像的一个带限逼近:

$$\begin{aligned} f_w(r, \alpha) &= \int_0^{2\pi} \int_{-1/2\Delta s}^{1/2\Delta s} G(R, \theta) \cdot W(R) \cdot e^{j2\pi Rr \cos(\alpha - \theta)} \cdot R dR d\theta \\ &= \int_0^{2\pi} \int_{-1}^1 g(s, \theta) \cdot h[\cos(\alpha - \theta) - s] ds d\theta \end{aligned} \quad (5-25)$$

在实际应用中主要应用的是上述公式离散化的形式。为简化计算, 卷积反投影重建公式在离散域的计算也是分步完成的: 先对  $M^{-1} \leq m \leq M^{+}$  计算出 1 个离散卷积  $g'_c()$ , 然后再通过插值计

算出插值结果  $g'_i()$  并最终计算得出重建图像。此过程数学描述形式如下:

$$g'_c(m\Delta s, n\Delta s) \approx \Delta s \sum_{m=-M}^{M^*} g(m\Delta s, n\Delta \theta) \cdot h[(m' - m)\Delta s] \quad (5-26)$$

$$g'_i(s', n\Delta \theta) \approx \Delta s \sum_{n=1}^N g'_c(m\Delta s, n\Delta \theta) \cdot I(s' - m\Delta s) \quad (5-27)$$

$$f_w(k\Delta x, l\Delta y) \approx \Delta \theta \sum_{n=1}^N g'_i(s', n\Delta \theta) \quad (5-28)$$

卷积反投影重建方法虽然和傅里叶反投影重建方法都源自傅里叶投影定理,但卷积反投影重建方法在应用范围上远比傅里叶反投影重建方法广泛得多。从实现算法上可以看出,卷积反投影重建方法无论是由软件实现还是由硬件实现,都是比较方便的。而且如果实验设计合理,在获取了高质量投影数据的前提下用卷积反投影重建方法可以得到比较清晰的重建图像。卷积反投影重建方法的以上几个特点在进行重建处理时具有明显优势。

### 3. 代数重建

代数重建技术 (Algebra Reconstruction Technique, ART) 也叫做迭代算法、优化技术等,是一种级数展开的重建方法。这种重建方法从一开始就在离散域中进行离散化分析,并可以直接得到数值解。由于其所需计算量要比变换法小得多,因此在处理时间上比较有优势。另外,此类级数展开重建方法重建出来的图像一般对比度都比较高,而且对于投影次数少的情况,也可以通过多次迭代将图像重建出来。由此可见,代数重建方法的实用性还是比较好的。

如图 5-7 所示,重建的目标放在一个直角坐标系网格中,网格各个像素按扫描次序从 1 排到  $N$ 。对于第  $j$  个像素,假设其吸收射线的能力为常量  $x_j$ ,那么第  $i$  条射线(由点放射源发出的射线共  $M$  条)与该像素的相交长度代表了这个像素在射线上对射线作用的权值。

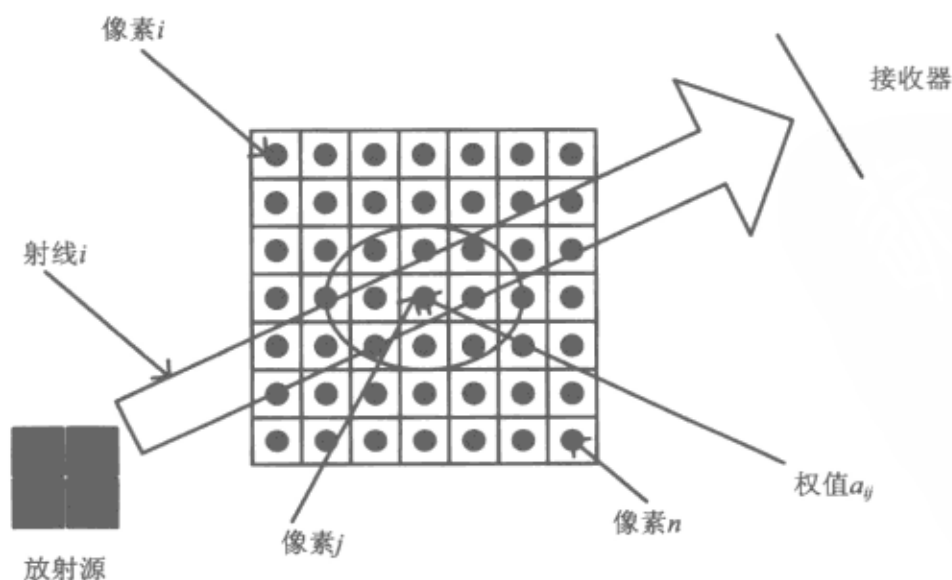


图 5-7 代数重建方法示意图

如果将射线沿此方向被物体吸收的总能量值记为  $y_i$ , 那么存在关系:

$$y_i \approx \sum_{j=1}^N x_j a_{ij} \quad (i=1, 2, \dots, M) \quad (5-29)$$

可以看出,  $M$  和  $N$  的取值将会影响重建图像的质量, 一般情况下它们均在 10 万量级之上。对于式 (5-29) 中的第  $i$  个方程, 其解集将构成一个超平面, 其法向量为  $(a_{ij})_{j=1}^N$ , 当前图像矢量  $\mathbf{x}^{(k)}$  在此超平面上的投影  $\mathbf{x}^{(k+1)}$  可用下式迭代得出:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \frac{y_i - (a_{ij})_{j=1}^N \cdot \mathbf{x}^{(k)}}{\|(a_{ij})_{j=1}^N\|^2} (a_{ij})_{j=1}^N \quad (5-30)$$

如果整个方程组存在唯一解, 那么由各个方程解集所构成的超平面将具备一个公共的交点, 最终的投影结果也将收敛于此。这样处理就将重建的问题转换成解线性方程组的问题, 在处理不完整投影时, 缺少投影数据等价于缺少方程式, 可按求解欠定方程组处理, 因此可以搁置数据不完整的问题并不影响对图像的重建。

#### 4. 超分辨率重建

1964 年, Harris 奠定了超分辨率得以存在的数学基础, 他将带限信号外推的方法对偶地运用到光学图像的超分辨重建之中, 取得了比较好的效果。80 年代初期, Tsai 和 Huang 首先提出了基于序列或多帧图像的超分辨重建问题, 并给出了基于频域逼近的图像重建方法。随着 IT 技术以及优化理论的发展, 人们在超分辨的研究上取得了突破性进展。早期的工作主要集中在频率域进行, 但随着对更一般的分辨率退化模型的考虑, 当前的研究工作几乎都集中在空间域进行。

频率域方法实际上是在频域内解决图像内插问题, 其观察模型基于傅里叶变换的移位特性, 具有理论简单、运算复杂度低、容易实现并行处理、可直观地去变形机制等优点。但这类方法基于的理论前提过于理想化, 不能有效地应用于多数场合, 只能局限于全局平移运动和线性空间不变降质模型, 包含空域先验知识的能力有限, 所以频域方法不是目前研究的主流。此外, Elad 和 Feuer 提出的一种通用的时域模型, 比较全面地考虑了成像中的各种实际情况, 这种方法的成像模型如图 5-8 所示。

空间域方法则认为低分辨率图像中的一个像素是由高分辨率图像的若干离散像元加权混合而成, 它将复杂的运动模型与相应的插值、迭代及滤波重采样放在一起进行处理, 涉及全局和局部运动、光学模糊、帧内运动模糊、空间可变点扩散函数、非理想采样及其他一些内容, 主要包括非均匀间隔样本内插法、迭代反向投影法、集合论 (如凸集投影: POCS) 方法、统计论 (如最大后验概率: MAP) 方法、混合 MAP/POCS 方法、正则化方法以及自适应滤波方法等。空域方法的适用范围较广, 具有很强的包含空域先验约束的能力, 例如马尔科夫随机场和凸集等先验约束, 这样在超分辨率重建过程中可以产生带宽外推, 但其运算量较大, 严重限制了这些方法的

使用。而非均匀样本内插方法、迭代反投影方法等，其结合先验信息的能力很弱，在改善超分辨率重建效果方面受到了极大的限制。

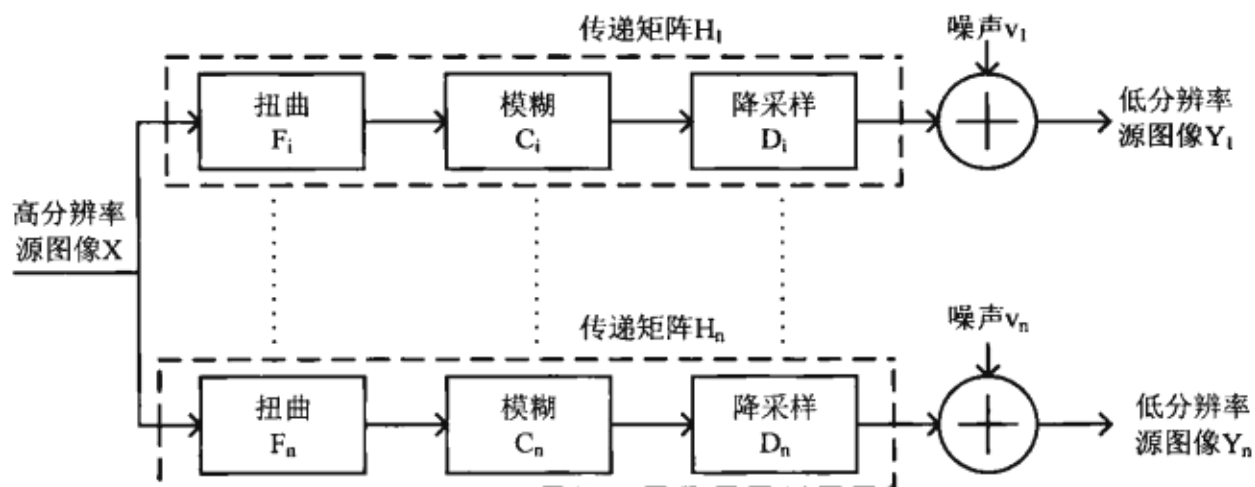


图 5-8 基于多帧的超分辨率图像重建的成像过程时域模型

## 5.2 系统功能

本章设计实现的 CT 图像重建系统主要完成人体头部切片的重建。

### 5.2.1 功能描述

CT 图像重建系统包含两个功能，第一是对圆锥体进行图像重建，主要是测试和校验读者在 VTK 和 CMake 安装和配置部分是否成功。第二是对人体头部的切片进行 CT 图像重建，将人体头部的三维重建效果展现出来。

#### 1. 圆锥体图像重建

CT 重建系统中的圆锥体图像重建是建立在 VTK 工具包基础之上的，所以这一部分主要以测试 VTK 的安装和配置为主，该部分系统的功能描述如下：

- 1) 创建圆锥体对象。通过调用 VTK 中的 ConeSource 类创建一个新的圆锥体源对象。
- 2) 配置圆锥体对象。对圆锥体对象的高、半径和精细度进行初始化。
- 3) 实现映射对象。创建一个新的映射对象，将圆锥体源对象关联到一个数据映射对象上。
- 4) 实现演员对象。创建一个演员对象，将数据映射对象关联到演员对象。
- 5) 实现演示者对象。创建一个演示者对象，将演员对象添加到演示者对象之中，并对演示者对象的背景色进行设置。
- 6) 创建绘制窗口。将演示者对象添加到绘制窗口对象之中并对绘制窗口进行大小的设置。
- 7) 绘制。通过不同角度进行圆锥体的绘制，实现不同视角观察圆锥体的功能。

## 2. 头部切片图像重建

头部切片图像重建主要处理的是一个人体头部的切片数据, 共有 93 个切片, 切片的间距是 1.5mm, 每个切片由间距为 3.2 毫米的  $64 \times 64$  像素构成。系统将由这些切片数据恢复出皮肤和骨骼的表面。具体系统功能描述如下:

1) 数据读入与检查。检查参数个数和参数值, 对图像数据进行读取。

2) 实现 2D 数据读者对象。创建 2D 数据读者对象, 设置数据的各维大小、字节排列方式、文件前缀、读取范围和 CT 数据切片间距和间隔。

3) 皮肤重建。采用 Marching Cubes 算法生成一个皮肤对象, 获取所读取的 CT 数据, 分别创建三角带对象和皮肤演员对象以及二者相应的映射关系对象。获得皮肤几何属性并设置皮肤颜色、反射率、反光强度和不透明度属性。

4) 骨骼重建。采用 Marching Cubes 算法生成一个骨骼对象, 获取所读取的 CT 数据, 分别创建三角带对象和骨骼演员对象以及二者相应的映射关系对象。获得骨骼几何属性并设置皮肤颜色、反射率、反光强度和不透明度属性。

5) 建立轮廓。在生成图像的四周实现轮廓的建立。

6) 建立相机。创建相机对象并设置相机对象的视角、位置和焦点值。

7) 实现演示。创建演示对象和演示窗口, 分别将轮廓、皮肤和骨骼放入到演示对象中以实现映射联系。实现相机在演示中的使用功能并调整窗口大小和相机演示截取平面。

8) 演示结束后实现对内存空间释放的功能。

### 5.2.2 界面效果

圆锥体 CT 图像重建结果如图 5-9 所示。

头像切片 CT 图像重建系统结果如图 5-10 所示。

头部切片 CT 图像重建系统结果俯视图如图 5-11 所示。

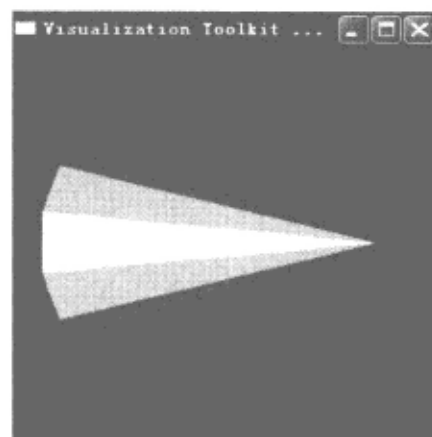


图 5-9 圆锥体 CT 图像重建结果图



图 5-10 头像切片 CT 图像重建系统结果图

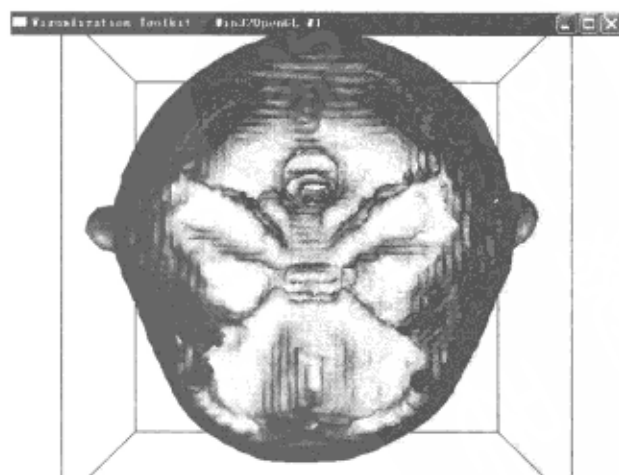


图 5-11 头像切片 CT 图像重建系统俯视图

## 5.3 系统结构与流程

CT 图像重建系统包含了圆锥体的图像重建和脑部切片的图像重建，主要使用了 VTK 提供的工具包，下面分别具体阐述 CT 图像重建系统的总体结构与主要流程。

### 5.3.1 总体结构

CT 图像重建系统通常要进行断层扫描、生成断层图像、预处理、分割断层组织和将新的断层组织切片重组或插值等操作。CT 图像重建系统总体结构如图 5-12 所示。

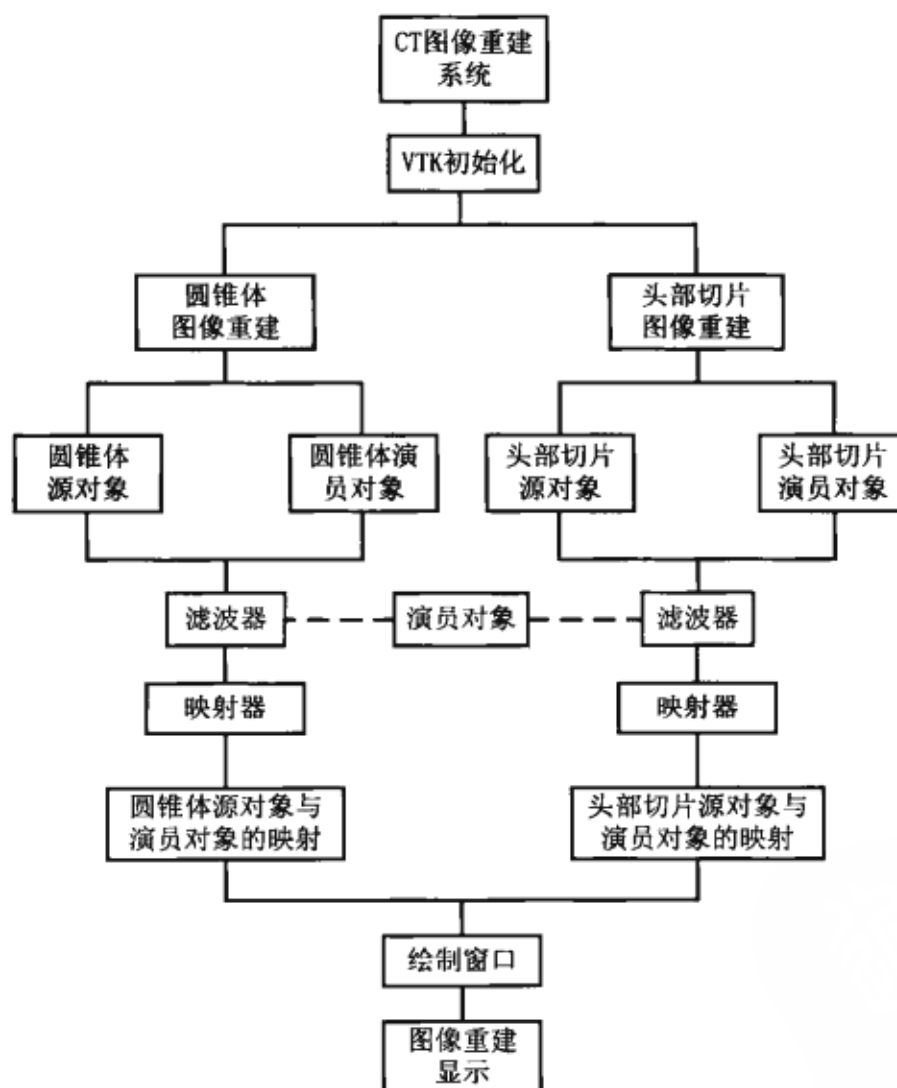


图 5-12 CT 图像重建系统总体结构图

### 5.3.2 主要流程

CT 图像重建系统包含了圆锥体 CT 图像重建和头部切片 CT 图像重建，系统中用到了 VTK 工具包的部分功能，这里也给出 VTK 工具包运行时的流程图。



数字图像处理典型案例详解

1. VTK 主要工作流程

VTK 工具包在 CT 图像重建系统调用时都会启动，其工作流程如图 5-13 所示。

2. 圆锥体 CT 图像重建主要流程

CT 图像重建系统中圆锥体图像重建主要包含了圆锥体图像模型的断层扫描、预处理、分割、切片重组、插值和三维重建显示的功能，根据这个特点，图 5-14 显示了圆锥体图像重建的主要流程。

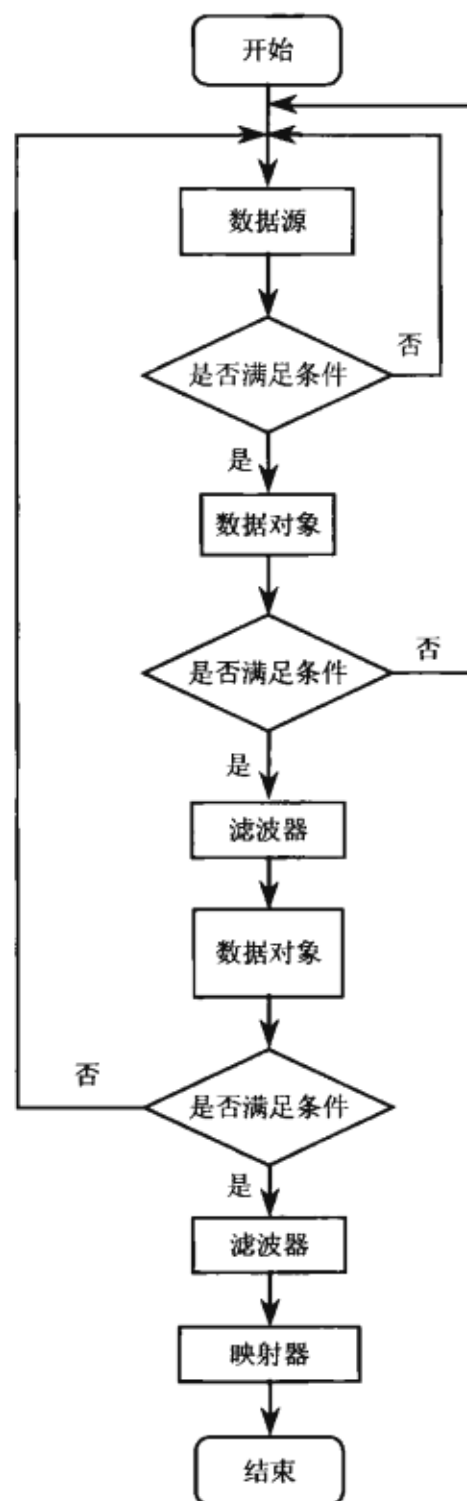


图 5-13 VTK 工作流程图

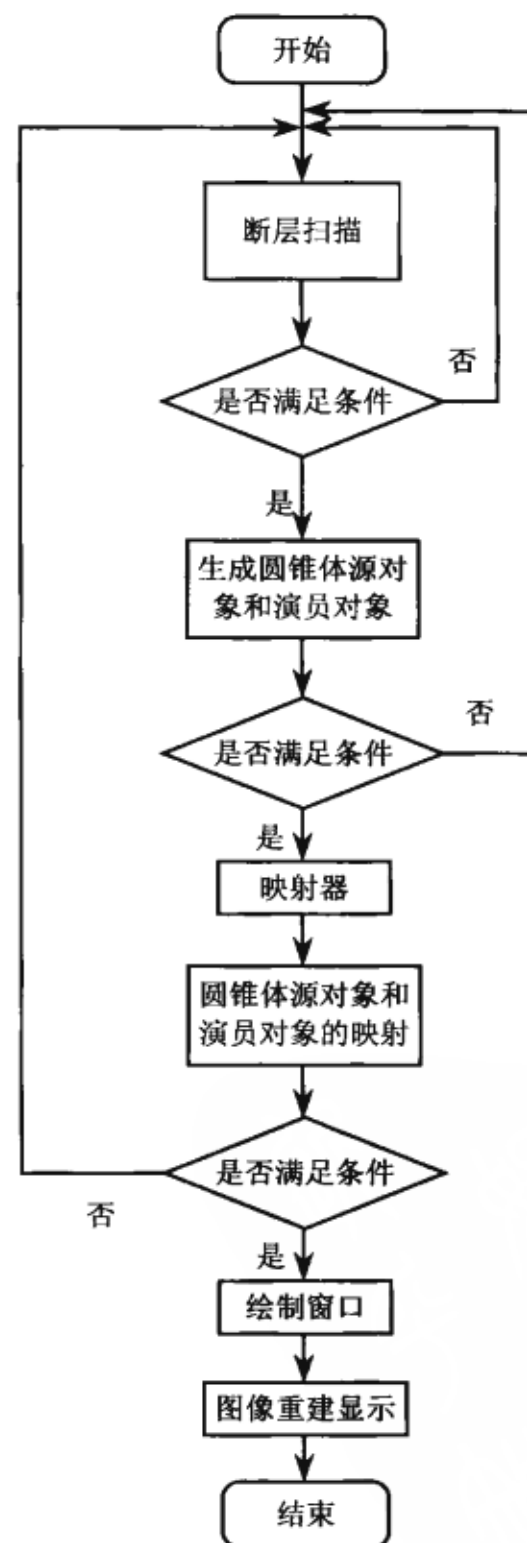


图 5-14 圆锥体图像重建流程图

### 3. 头部切片 CT 图像重建主要流程

CT 图像重建系统中头部切片图像重建主要包含了头部切片图像模型的断层扫描、预处理、分割、切片重组、插值和三维重建显示的功能，图 5-15 显示了其主要流程。

### 4. CT 图像重建系统总流程图

CT 图像重建系统包含了圆锥体和脑部切片的断层扫描、预处理、分割、切片重组、插值和三维重建显示的功能。系统总流程图如图 5-16 所示。

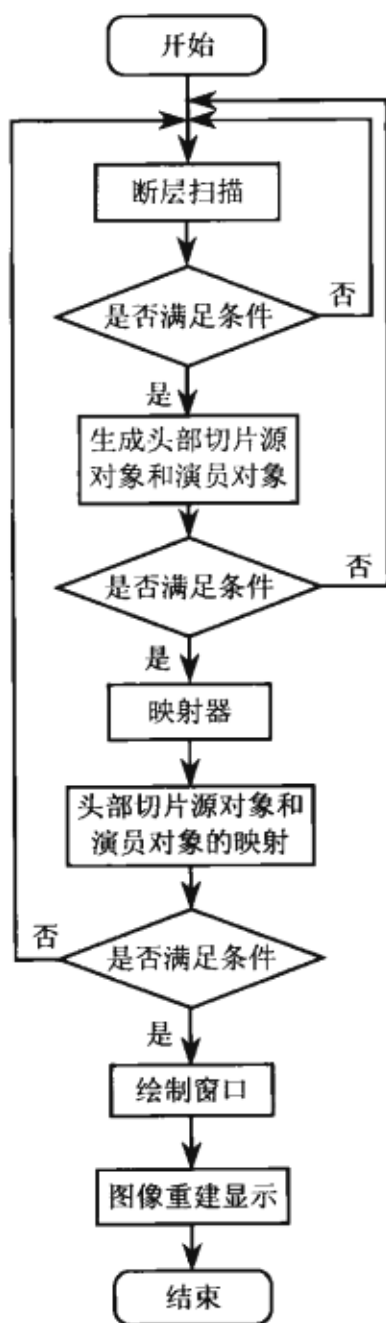


图 5-15 头部切片图像重建流程图

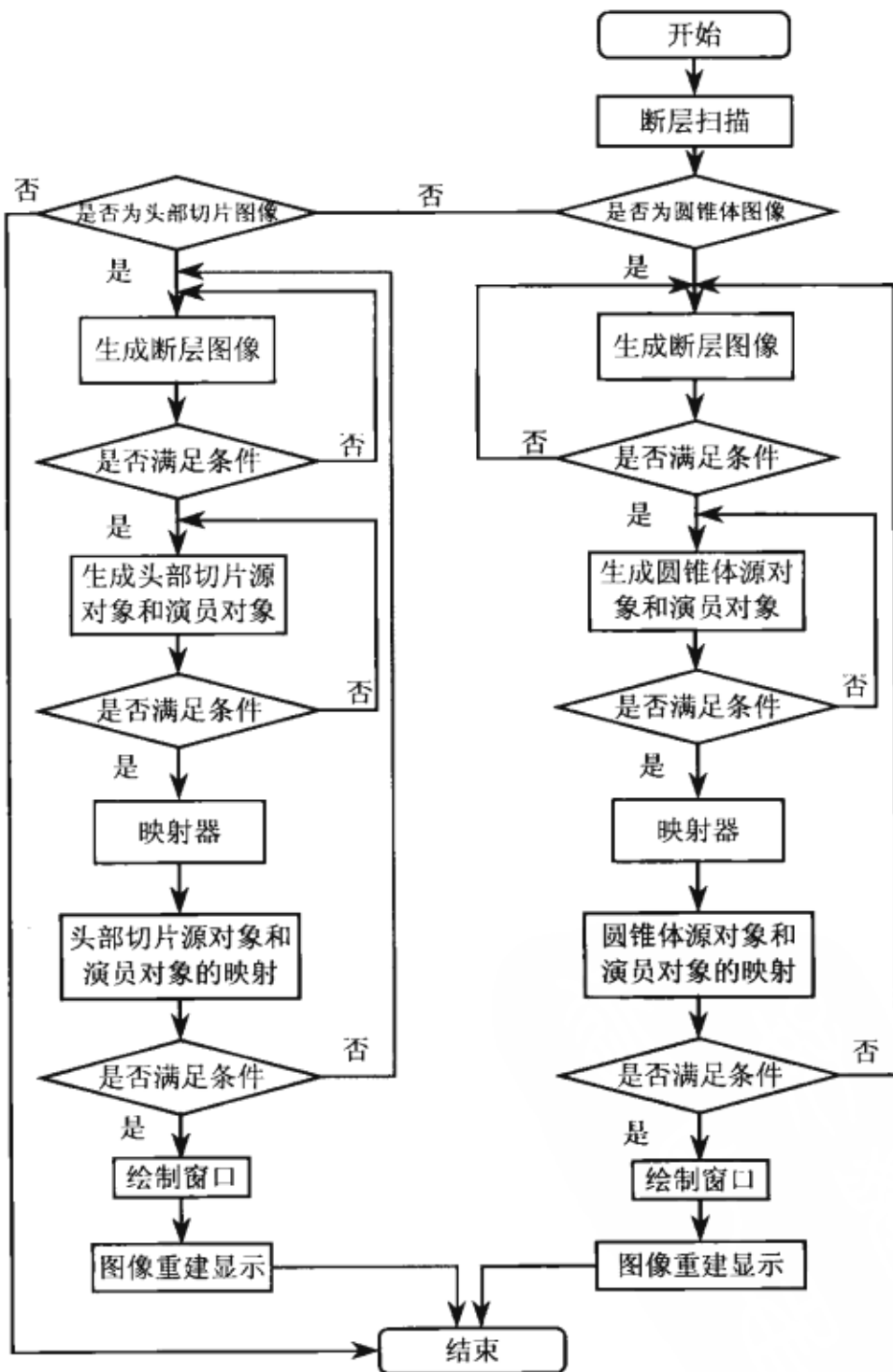


图 5-16 CT 图像重建系统总流程图

## 5.4 编程实现

CT 图像重建系统采用 VC 2008 开发平台结合 VTK 编程实现。在重建的过程中,需要用到两部分重建图像数据。首先,在圆锥体 CT 图像重建系统中用到的图像重建数据是在 VTK 软件中已经封装好的圆锥体函数 cone,由于这部分数据已经封装成为固定自带函数,所以在调用之后即可运行出重建结果。第二部分是头部切片 CT 重建系统,这一部分的数据并不是 VTK 软件自身所带,而是在 VTK 官方网站上的 VTKDATA 数据素材包中,VTKDATA 数据素材包中包含了支持 VTK、C++和 Java 等多种编程语言的重建素材,头部切片 CT 重建系统中采用的数据素材位于 VTKDATA 的 headsq 文件中,其中包含了从 quarter.1 到 quarter.93 共 93 个头部切片素材,头部切片重建系统则是通过利用其中的全部素材数据进行重建。

### 5.4.1 圆锥体 CT 图像重建系统

该部分主要是对 VTK 的配置进行测试,其代码如下:

```
#include "vtkConeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkCamera.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include "vtkProperty.h"
#include "Windows.h"

int main()
{
    // 创建一个圆锥体源对象
    vtkConeSource *cone = vtkConeSource::New();
    // 设置圆锥体的高
    cone->SetHeight( 3.0 );
    // 设置圆锥体的半径
    cone->SetRadius( 1.0 );
    // 设置圆锥体的精细度
    cone->SetResolution( 10 );

    // 创建一个数据映射对象
    vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
    // 将圆锥体源对象关联到数据映射对象
    coneMapper->SetInputConnection( cone->GetOutputPort() );

    // 创建一个演员对象
    vtkActor *coneActor = vtkActor::New();
```

```

// 将数据映射对象关联到演员对象
coneActor->SetMapper( coneMapper );

// 创建一个演示者对象
vtkRenderer *aRenderer= vtkRenderer::New();
// 将演员对象添加到演示者对象
aRenderer->AddActor( coneActor );
// 设置演示者对象的背景颜色
aRenderer->SetBackground( 0.1, 0.2, 0.4 );

// 创建一个绘制窗口对象
vtkRenderWindow *renWin = vtkRenderWindow::New();
// 将演示者对象添加到绘制窗口中
renWin->AddRenderer( aRenderer );
// 设置绘制窗口大小
renWin->SetSize( 300, 300 );

// 从不同角度观察圆锥体
for (int i = 0; i < 25; ++i)
{
    // 绘制
    renWin->Render();
    // 暂停
    Sleep(500);
    // 改变视角
    aRenderer->GetActiveCamera()->Azimuth( i );
}

// 清除对象, 释放空间
cone->Delete();
coneMapper->Delete();
coneActor->Delete();
aRenderer->Delete();
renWin->Delete();

return 0;
}

```

在此类中, 首先是调用 VTK 中的 `vtkConeSource` 创建一个圆锥体源对象, 再对该圆锥体的一个对象 `cone` 设置其高、半径和精细度。然后再调用 `vtkPolyDataMapper` 方法创建一个数据映射对象 `coneMapper`, 并将这个圆锥体源对象关联到数据映射对象。接下来利用 `vtkActor` 创建一个演员对象 `coneActor`, 再将这个对象通过数据映射跟演员对象关联。下一步通过 `vtkRenderer` 类创建一个 `aRenderer` 演示者对象, 这个对象的作用是将圆锥体显示在屏幕上, 通过调用 `vtkRenderer` 中的 `AddActor` 函数将演员对象载入其中, 这里主要将 `coneActor` 参数传入 `AddActor` 函数中。之后设置

下演示者对象的背景颜色以完成整个演示者对象的操作。接下来通过 `vtkRenderWindow` 方法创建一个窗口, 这里注意创建的窗口指向的是窗口对象的指针而不是一个窗口对象。调用 `renWin` 对象中的 `AddRenderer` 函数将对象 `aRenderer` 载入其中。当然, 这里同样可以对窗口的大小进行设置。此模型是一个三维全角度可视化的模型, 所以需要对其进行全方位的观察, 在 `aRenderer` 中调用 `GetActiveCamera()` 函数中的 `Azimuth()` 方法, 即可实现该功能。全部操作完毕之后需要对预先创建的圆锥体对象 `cone`、数据映射对象 `coneMapper`、演员对象 `coneActor`、演示者对象 `aRenderer` 和窗口对象 `renWin` 进行内存的释放。

在完成以上代码之后, 打开项目属性对话框, 并在【链接器】/【输入】/【附加依赖项】中加入如下库文件, 库文件之间使用空格分隔。

```
vtkViews.lib vtkInfovis.lib vtkRendering.lib vtkImaging.lib vtkIO.lib vtkWidgets.lib
vtkHybrid.lib vtkRendering.lib vtkIO.lib vtkexoIIC.lib vtklibxml2.lib vtkalglib.lib
vtkGraphics.lib
vtkverdict.lib vtkftgl.lib vtkfreetype.lib opengl32.lib vtkFiltering.lib
vtkCommon.lib
vtkDICOMParser.lib vtkNetCDF.lib vtkmetaio.lib comctl32.lib wsock32.lib
vtksqlite.lib
vtkpng.lib vktiff.lib vtkzlib.lib vtkjpeg.lib vtkexpat.lib vtksys.lib ws2_32.lib
vfw32.lib
```

完成库文件的操作之后, 单击工具栏中的  按钮运行程序即可。

### 5.4.2 头部切片 CT 图像重建系统

头部切片 CT 图像重建系统主要是对一个头部的切片进行断层扫描、预处理、分割、切片重组和插值从而实现三维重建的图像重建系统, 其代码如下:

```
//=====
// 基于 MC 算法的 CT 图像重建
//=====

#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkVolume16Reader.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkOutlineFilter.h"
#include "vtkCamera.h"
#include "vtkPolyDataMapper.h"
#include "vtkStripper.h"
#include "vtkPolyDataNormals.h"
```

```

#include "vtkMarchingCubes.h"

//-----
// 主函数
//-----
int main (int argc, char **argv)
{
    // 参数个数检查
    if(argc < 2)
    {
        cout << "用法: " << argv[0] << " DATADIR/headsq/quarter" << endl;
        return 1;
    }

    // -----读取数据-----

    // 创建 D 数据读者对象
    vtkVolume16Reader *v16 = vtkVolume16Reader::New();
    // 设置数据的各维大小
    v16->SetDataDimensions(64,64);
    // 设置数据的字节排列方式
    v16->SetDataByteOrderToLittleEndian();
    // 设置文件前缀, 它将结合下一参数, 以 "FilePrefix.%d" 方式读取随后文件
    v16->SetFilePrefix (argv[1]);
    // 设置文件读取范围
    v16->SetImageRange(1, 93);
    // 设置 CT 数据的切片间距和间隔
    v16->SetDataSpacing (3.2, 3.2, 1.5);

    // -----重建皮肤-----

    // 创建一个 Marching Cubes 算法的对象
    vtkMarchingCubes *skinExtractor = vtkMarchingCubes::New();
    // 获得所读取的 CT 数据
    skinExtractor->SetInputConnection(v16->GetOutputPort());
    // 提取出 CT 值为的皮肤数据
    skinExtractor->SetValue(0, 500);
    // 重新计算法向量
    vtkPolyDataNormals *skinNormals = vtkPolyDataNormals::New();
    skinNormals->SetInputConnection(skinExtractor->GetOutputPort());
    skinNormals->SetFeatureAngle(60.0);
    // 创建三角带对象
    vtkStripper *skinStripper = vtkStripper::New();
    // 将生成的三角片连接成三角带
    skinStripper->SetInputConnection(skinNormals->GetOutputPort());
    // 创建一个数据映射对象

```



```
vtkPolyDataMapper *skinMapper = vtkPolyDataMapper::New();
// 将三角带映射为几何数据
skinMapper->SetInputConnection(skinStripper->GetOutputPort());
skinMapper->ScalarVisibilityOff();
// 创建一个代表皮肤的演员对象
vtkActor *skin = vtkActor::New();
// 获得皮肤几何数据的属性
skin->SetMapper(skinMapper);
// 设置皮肤颜色的属性
skin->GetProperty()->SetDiffuseColor(1, .49, .25);
// 设置反射率
skin->GetProperty()->SetSpecular(.3);
// 设置反射光强度
skin->GetProperty()->SetSpecularPower(20);
// 设置不透明度
skin->GetProperty()->SetOpacity(1.0);

// -----重建骨骼-----

// 创建一个 Marching Cubes 算法的对象
vtkMarchingCubes *boneExtractor = vtkMarchingCubes::New();
// 获得所读取的 CT 数据
boneExtractor->SetInputConnection(vl6->GetOutputPort());
// 提取出 CT 值为骨骼的数据
boneExtractor->SetValue(0, 1150);
// 重新计算法向量
vtkPolyDataNormals *boneNormals = vtkPolyDataNormals::New();
boneNormals->SetInputConnection(boneExtractor->GetOutputPort());
boneNormals->SetFeatureAngle(60.0);
// 创建三角带对象
vtkStripper *boneStripper = vtkStripper::New();
// 将生成的三角片连接成三角带
boneStripper->SetInputConnection(boneNormals->GetOutputPort());
// 创建一个数据映射对象
vtkPolyDataMapper *boneMapper = vtkPolyDataMapper::New();
// 将三角带映射为几何数据
boneMapper->SetInputConnection(boneStripper->GetOutputPort());
boneMapper->ScalarVisibilityOff();
// 创建一个代表骨骼的演员对象
vtkActor *bone = vtkActor::New();
// 获得骨骼几何数据的属性
bone->SetMapper(boneMapper);
// 设置骨骼颜色的属性
bone->GetProperty()->SetDiffuseColor(1, 1, .9412);

// -----建立轮廓-----
```

```
// 在生成图像的四周建立轮廓, 以方便与用户交互
vtkOutlineFilter *outlineData = vtkOutlineFilter::New();
outlineData->SetInputConnection(v16->GetOutputPort());
vtkPolyDataMapper *mapOutline = vtkPolyDataMapper::New();
mapOutline->SetInputConnection(outlineData->GetOutputPort());
vtkActor *outline = vtkActor::New();
outline->SetMapper(mapOutline);
outline->GetProperty()->SetColor(0,0,0);

// -----建立相机-----

// 创建一个相机对象
vtkCamera *aCamera = vtkCamera::New();
// 设置相机的视角
aCamera->SetViewUp (0, 0, -1);
// 设置相机的位置
aCamera->SetPosition (0, 1, 0);
// 设置相机的焦点
aCamera->SetFocalPoint (0, 0, 0);
aCamera->ComputeViewPlaneNormal();

// -----准备演示-----

// 创建演示者对象
vtkRenderer *aRenderer = vtkRenderer::New();
// 创建演示窗口对象
vtkRenderWindow *renWin = vtkRenderWindow::New();
// 将演示者添加到演示窗口
renWin->AddRenderer(aRenderer);
// 创建演示窗口交互对象, 使得用户可以用鼠标、键盘和窗口交互
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
// 将演示窗口关联到演示窗口交互对象
iren->SetRenderWindow(renWin);

// 将轮廓添加到演示者对象
aRenderer->AddActor(outline);
// 将皮肤添加到演示者对象
aRenderer->AddActor(skin);
// 将骨骼添加到演示者对象
aRenderer->AddActor(bone);
// 设置演示者所使用的相机
aRenderer->SetActiveCamera(aCamera);
// 使相机生效
aRenderer->ResetCamera();
aCamera->Dolly(1.5);
// 设置演示者的背景颜色
aRenderer->SetBackground(1,1,1);
```

```

// 调整相机截取平面
aRenderer->ResetCameraClippingRange();

// 设置演示窗口大小
renWin->SetSize(640, 480);

// -----开始演示-----

// 演示窗口交互对象初始化
iren->Initialize();
// 开始演示
iren->Start();

// -----演示结束-----

// 删除对象, 释放空间
v16->Delete();
skinExtractor->Delete();
skinNormals->Delete();
skinStripper->Delete();
skinMapper->Delete();
skin->Delete();
boneExtractor->Delete();
boneNormals->Delete();
boneStripper->Delete();
boneMapper->Delete();
bone->Delete();
outlineData->Delete();
mapOutline->Delete();
outline->Delete();
aCamera->Delete();
aRenderer->Delete();
renWin->Delete();
iren->Delete();

return 0;
}

```

上段代码中 CT 图像重建的主要方法是采用 MC 算法进行重建。首先是创建一个 2D 的数据读者对象, 这里调用了 `vtkVolume16Reader` 对象, 创建一个 `v16` 的实例, 需要注意的是创建的 `v16` 是一个指向该实例的指针。接下来调用该对象中的 `SetDataDimensions` 方法, 这个方法的作用是设置数据的各个维度大小, 这里为  $64 \times 64$ 。接着设置数据的字节排列方式, 可以通过调用 `SetDataByteOrderToLittleEndian` 方法实现。为了使一些函数能正常读入文件, 需要对文件的前缀进行改动, 所以调用 `SetFilePrefix` 函数设置好文件的参数, 使得系统可以以 `FilePrefix.%d` 的形式

读取后面的文件。由于读入的图像文件大小不能没有限制，所以这里调用 `SetImageRange` 函数对文件的大小进行限制，程序中的限制是从 1 到 93，在产生图像数据的切片时，需要对其间隔和间距进行特殊的规定，所以这里通过 `SetDataSpacing` 方法对切片的间距进行设置。

在对文件进行读取之后，通过该文件对头部的皮肤进行重建。首先通过 `vtkMarchingCubes` 类生成一个新的 `skinExtractor` 对象，这个对象是用于皮肤重建的对象，然后调用 `skinExtractor` 对象中的 `SetInputConnection` 方法获取文件中的 CT 数据，在 `SetValue` 方法中，可以提取出所有 CT 值为 500 的皮肤数据。下一步要计算出每一个部分的法向量，这里主要是通过 `vtkPolyDataNormals` 类实现，主要调用该类中的 `SetInputConnection` 方法和 `SetFeatureAngle` 方法。这样就可以将取出的 CT 数据转变为向量值。接下来用 `vtkStripper` 类创建三角带对象，同样仅取出三角片的值，接着调用 `SetInputConnection` 方法将三角片的值连接成三角带。在完成三角带创建之后，要将三角带映射为几何数据，所以要通过 `vtkPolyDataMapper` 创建一个数据映射对象 `skinMapper`，之后调用 `SetInputConnection` 方法和 `ScalarVisibilityOff` 方法将三角带的几何数据完成。几何数据创建完毕后即可将皮肤显示在界面中，这里同样需要利用 `vtkActor` 创建一个皮肤的演员对象 `skin` 并完成 `skin` 与 `skinMapper` 之间的映射，接下来可以通过调用 `skin` 对象中 `GetProperty` 方法下的 `SetDiffuseColor`、`SetSpecular`、`SetSpecularPower` 和 `SetOpacity` 对该皮肤对象进行颜色属性、反射率、反射光强度和不透明度的设置。

皮肤重建完毕之后是骨骼的重建，首先通过 `vtkMarchingCubes` 类生成一个新的 `boneExtractor` 对象，这个对象是要对其进行骨骼重建的对象，然后调用 `boneExtractor` 对象中的 `SetInputConnection` 方法获取文件中的 CT 数据，在 `SetValue` 方法中，可以提取出所有 CT 值为 1150 的骨骼数据。然后计算出每一个部分的法向量，这里主要是通过 `vtkPolyDataNormals` 类实现，主要调用该类中的 `SetInputConnection` 方法和 `SetFeatureAngle` 方法。这样就可以将取出的 CT 数据转变为向量值。接下来是用 `vtkStripper` 类创建三角带对象，同样仅取出三角片的值，接着调用 `SetInputConnection` 方法将三角片的值连接成三角带。在完成三角带创建之后，要将三角带映射为几何数据，所以要通过 `vtkPolyDataMapper` 创建一个数据映射对象 `boneMapper`，之后调用 `SetInputConnection` 方法和 `ScalarVisibilityOff` 方法将三角带的几何数据完成。几何数据创建完毕后即可将骨骼显示在界面中，这里同样需要利用 `vtkActor` 创建一个骨骼的演员对象 `bone` 并完成 `bone` 与 `boneMapper` 之间的映射，接下来可以通过调用 `bone` 对象中 `GetProperty` 方法下的 `SetDiffuseColor` 对该骨骼对象进行颜色属性的设置。

骨骼重建完毕之后要建立能将三维脑部头像融入其中的轮廓，为了生成四周的轮廓，需要通过 `vtkOutlineFilter` 类创建一个新的对象 `outlineData`，通过调用其中的 `SetInputConnection` 对 CT 文件进行读取，生成新的映射对象 `mapOutline`，将数据和该映射对象进行关联完成轮廓的建立，之后调用 `vtkActor` 生成一个轮廓演员，分别将映射对象传入其中并通过调用 `SetColor` 方法对轮廓颜

色进行设置。

此时所有要演示的部分都已经创建和设置完毕，接下来创建相机和准备演示部分。创建相机主要是调用 `vtkCamera` 类，新建一个 `aCamera` 对象，之后调用其中的 `SetViewUp`、`SetPosition`、`SetFocalPoint` 和 `ComputeViewPlaneNormal` 这 4 个方法分别设置相机的视角、位置和焦点。在演示之前，要对演示部分进行预处理以做好准备演示的功能，首先是利用 `vtkRenderer` 类创建一个 `Arnderer` 对象，然后利用 `vtkRenderWindow` 类创建一个 `renWin` 窗口对象，之后调用 `renWin` 对象中 `AddRenderer` 方法将演示者添加到演示窗口之中。下面使用 `vtkRenderWindowInteractor` 类创建一个 `iren` 窗口交互对象，使得用户可以通过鼠标和键盘等外部设备与窗口进行交互。在创建了 `iren` 之后，需要将 `renWin` 通过 `SetRenderWindow` 方法以参数的形式传入其中。之后利用 `aRenderer` 对象中的 `AddActor` 方法分别将 `outline`、`skin` 和 `bone` 以参数的形式传入，从而分别实现将轮廓、皮肤和骨骼添加到演示者对象之中。加载完毕之后再对相机进行初始化，为了使相机工作，将上文中的 `aCamera` 对象通过 `SetActiveCamera` 函数将其激活生效。由于每次读入切片都要重置相机，所以要对相机进行重启操作，即调用 `ResetCamera` 和 `Dolly` 函数。通过调用 `SetBackground` 函数可以对演示者进行背景颜色的设置，再通过 `ResetCameraClippingRange` 方法对相机进行平面截取，之后再对演示窗口进行大小的设置即可，这里的大小设置并不是固定的，其中传入的参数分别是横轴与纵轴的长度，根据切片图像大小而定。

在开始演示的环节中，只需使用 `iren` 对象调用其中的 `Initialize` 方法即可，之后调用 `Start` 函数进行图像的演示。

在演示结束后要依次对 `v16`、`skinExtractor`、`skinNormals`、`skinStripper`、`skinMapper`、`skin`、`boneExtractor`、`boneNormals`、`boneMapper`、`boneStripper`、`bone`、`outlineData`、`mapOutline`、`outline`、`aCamera`、`aRenderer`、`renWin` 和 `iren` 调用 `Delete` 方法进行内存的释放。

程序代码无误之后对库文件进行操作，打开项目属性对话框，并在【链接器】/【输入】/【附加依赖项】中加入如下库文件，库文件之间使用空格分隔：

```
$(NOINHERIT) kernel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib
oleaut32.lib
uuid.lib comdlg32.lib advapi32.lib vtkRendering.lib vtkIO.lib vtkDICOMParser.lib
vtkNetCDF.lib vtkmetaio.lib comctl32.lib vtksqlite.lib vtkpng.lib vtktiff.lib
vtkzlib.lib
vtkjpeg.lib vtkexpat.lib vfw32.lib vtkGraphics.lib vtkverdict.lib vtkImaging.lib
vtkFiltering.lib
vtkCommon.lib vtksys.lib ws2_32.lib wsock32.lib vtkftgl.lib vtkfreetype.lib
opengl32.lib
```

导入库文件后按 F7 键或选择菜单【生成】/【生成解决方案】，生成解决方案。在项目所在目录下的 `Debug` 目录中，找到生成的 `CTImageReconstruction_brain` 程序，将其复制到



VTKData\Data\headsq 目录下。打开命令行窗口，将目录定位到 VTKData\Data\headsq。在命令行窗口中输入：CTImageReconstruction\_brain quarter，回车即可。

## 5.5 经验分享

本章主要讲述了 CT 图像重建系统的核心原理、功能、流程、扫描和重建的一般方法。在编程实现阶段中分别用了圆锥体的图像重建系统和头部切片重建系统两个程序为读者讲解并展示了如何实现 CT 图像重建。从整体角度看，CT 图像重建系统主要包括了断层扫描、生成断层图像、预处理、分割断层组织和将新的断层组织切片重组或插值等操作。从具体程序来看，在进行头部切片的 CT 重建时主要包含了皮肤重建、骨骼重建、轮廓创建以及演示三维重建图像。

从 CT 图像重建系统的鲁棒性角度考虑，系统在一定程度上考虑了鲁棒性，这点主要体现在对图像的预判断上，通过 SetFilePrefix 方法可以将要读入的文件统一进行格式处理。同样对不同切片的情况也予以了考虑并通过 SetDataSpacing 方法对切片之间的间距大小进行了严格的规定，从而使得系统从读入文件开始到文件预处理等阶段比较严谨，从系统的结果图中也不难发现整个系统的体系比较完整。

从 CT 图像重建系统的可靠性和可兼容性角度考虑，系统由于在每次生成图像之后都会对内存进行释放，并不会产生内存堆栈溢出等问题。另外一方面，由于程序采用的是 C++ 编译，所以可能会产生一些兼容性的问题上，比如本机的工作环境是 32 位，如果在 64 位的平台上运行时，个别变量会产生错误，这点也是需要改进的地方。

从 CT 图像重建系统的可扩展性角度考虑，相对较好，其中开放了大量的 VTK 接口，读者可以根据不同的需求将系统中的任意部分做成一个新的接口，方便今后使用。

从 CT 图像重建系统的易用性角度考虑，系统自身易用性较好，但是在 CMake 中对 VTK 的配置以及编译之后对系统文件的再次转移确实需要读者细心操作并按步完成，这也是系统欠缺的一部分，读者可以根据自身需要将编译生成的文件通过代码自动复制到指定文件夹中以提高系统的易用性和安全性。此外，对 VTK 进行配置的过程中，由于个人操作系统和机器配置不同，可能会出现各种各样的问题，如果配置完毕后圆锥体 CT 重建系统无法正常显示请勿继续做头部切片图像重建系统，应按照书中 VTK 配置部分仔细进行配置，配置时对于书中要求选择的选项务必配置正确，其他的额外选项在没有特殊需求的情况下并不建议勾选。



## 第 6 章 数字图像水印系统

“不要因为也许会改变，就不肯说那句美丽的誓言。不要因为也许会分离，就不敢求一次倾心的相遇。总有一些什么会留下来的吧，留下来作一件不灭的印记……”这印记是诗人席慕容笔下爱的见证。造纸时，通过改变纸浆纤维密度而在纸中留下一些迎光可视的隐藏印记，便是人们所熟知的水印，它可以用作真伪的见证。水印技术发端于唐代，最初只是为了增加纸张的艺术美感，从宋代开始用于纸币防伪。随着信息技术的发展，传统水印也突破纸上乾坤而跻身信息世界，成为数字水印。数字水印是一种隐藏在数字化图像、视频或音频等多种媒体中的信息，在版权保护、防伪认证、隐蔽通信等领域具有重要的应用价值。本章结合一个数字图像水印系统解读水印技术的基本原理和编程实现方法。

**本章要点：**

- 数字图像水印嵌入技术
- 数字图像水印提取技术
- 数字图像水印系统功能描述
- 数字图像水印系统的总体结构和主要流程
- 数字图像水印系统的编程实现

### 6.1 核心技术原理

目前，数字图像水印的研究主要可以分为理论基础研究、应用基础研究、应用技术研究三个层次。数字水印从分类的角度也分为空间域和交换域等多个方面。无论从研究方向还是从分类角度来看，数字图像水印系统彼此之间都是具有很大区别的，因此，研究数字图像水印的理论模型就显得尤为重要。在大多数情况下，数字图像水印模型都由水印生成嵌入系统和水印检测恢复系统所组成。

数字图像水印生成嵌入系统模型如图 6-1 所示，其中输入信息为载入数据、宿主数据以及可以选择的公钥或私钥组成。载入数据也就是水印数据可以是灰度图像或位图等任何形式的数字图像。

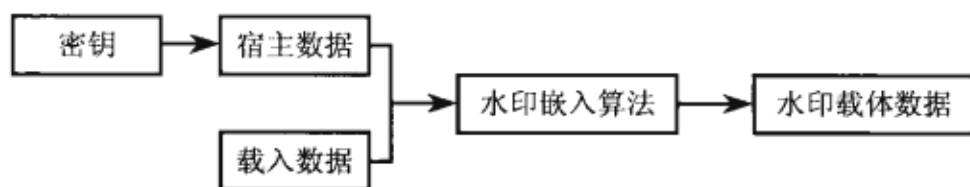


图 6-1 水印嵌入系统模型图

数字图像水印提取系统模型如图 6-2 所示，其中虚线部分的原始宿主数据表示为在进行水印的检测提取过程中是否需要原始宿主数据，该部分不是必要的。

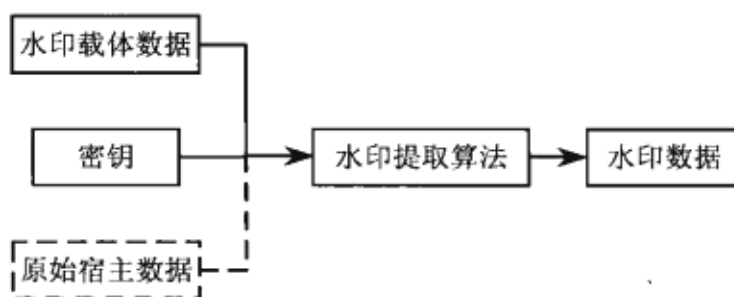


图 6-2 水印提取系统模型图

### 6.1.1 图像水印嵌入技术

在数字水印的生成阶段，嵌入算法和嵌入方案的主要目标是使数字水印在不可见性和鲁棒性之间找到一个较好的折中。根据所基于的域不同，数字水印嵌入技术主要分为时空域算法、变换域算法和压缩域算法三大类。时空域算法将水印信息直接嵌入到音频时域采样、图像空间像素和视频数据（按帧或者沿时间轴）等原始载体数据中，即在媒体信号的时间域或空间域上实现水印嵌入。变换域算法将水印信息嵌入到音频、图像、视频和三维目标等原始载体的变换域系数中。压缩域算法广义上是指充分考虑 JPEG、MPEG 和 VQ 技术的结构和特性，将水印嵌入到压缩过程的各种变量值域中，以提高对相应压缩技术或压缩标准攻击的鲁棒性为目标的嵌入算法。

#### 1. 空间域图像水印嵌入技术

在时空域算法中，重要的一大类算法是脆弱水印和半脆弱水印算法，因为这类算法具有对攻击的时间或空间位置的定位能力。因此，后期人们研究的时空域算法多用于内容认证或篡改提示。在时空域中，由于时域往往与音频水印嵌入算法结合紧密，这里暂不做过多阐述。在空间域中的图像水印嵌入技术主要包含加性和乘性、位平面、统计特征、替换、量化、关系、自适应等算法和半色调图像的水印嵌入算法等。这些算法彼此也互有交叉，在此并不一一列举，下面详细介绍 4 种较为典型的空间域下的嵌入算法，即拼凑法、利用黑白像素个数奇偶性法、最低有效位水印嵌入法，以及不带嵌入因子的简单加性算法。

##### （1）拼凑法

拼凑法（Patchwork）最初由 Bender 等人在 1996 年提出，他的主要思想是考虑到图像等载体

通常都具有一些统计特性，例如均值、方差等。因此，可以考虑利用载体的这些特性来携带水印信息。从本质上来说，拼凑法就是一种通过在原始载体信号中改变载体信号某种特定的统计特性来携带水印信息的方法。其主要步骤如下：

1) 随机选择两个集合  $A=\{a_i\}$  和  $B=\{b_i\}$ , 但是要求  $A$  和  $B$  中含有相同数量的图像系数, 设为  $n$ 。

2) 将集合  $A$  中的所有样值增加一个常量  $d$ , 同时将集合  $B$  中的所有样值减少同样的常量  $d$ 。这样两个集合中的样值都经过了微小的改动。用数学公式描述如下:

$$a_i^* = a_i + d, b_i^* = b_i - d, i = 1, 2, \dots, n, a_i \in A, b_i \in B \quad (6-1)$$

这里根据图像的横坐标与纵坐标之和的奇偶性不同将图像数据分为两组，在横坐标与纵坐标的值和为偶数的所有系数上增加常数  $d=2.8$ （常量值  $d$  并没有严格要求，只是作为一个举例），在横坐标与纵坐标之和为奇数的所有系数上减少常数  $d=2.8$ ，从而嵌入 1 比特信息。通过适当地调整参数，拼凑法对 JPEG 压缩、FIR 滤波以及图像裁剪有一定的抵抗力，但该方法嵌入的信息量有限。为了嵌入更多的水印信息，可以将图像分块，然后对每一个图像块进行嵌入操作。目前较被认可的是一种两层嵌入法，第一层水印在各块的直流分量上，第二层水印利用上述的拼凑方法，目的是进一步提高鲁棒性，属于多重水印嵌入技术。

此外, V.Ratnakar 提出一种零均值补丁的水印方法也是从这里衍生而出。其中每个补丁由个数相等的 1 和 -1 组成, 文中采用的补丁共有 8 种形式, 如图 6-3 所示。得到 8 种基本形式的补丁之后, 在每个补丁中随机选择少数几对像素, 每一对中的一个加 0.2, 另一个减 0.2, 以扩展成  $M$  个最终带嵌入的补丁。假设原始图像大小为  $N_1 \times N_2$ , 一共要嵌入  $M$  个大小为  $k \times k$  的补丁  $P = \{p_0, p_1, \dots, p_{M-1}\}$ 。在嵌入第  $i$  个补丁  $p_i$  时, 首先在原始载体图像中随机选择嵌入位置  $(u_i, v_i)$ ,  $0 \leq u_i < N_1 - k$ ,  $0 \leq v_i < N_2 - k$ 。然后计算该补丁与左上角坐标为  $(u_i, v_i)$ , 右下角坐标为  $(u_i + k - 1, v_i + k - 1)$  的图像块之间的相关值。如果这个相关值大于某个阈值, 则不修改图像。否则, 计算表征该图像区域复杂度的方差特征量, 令嵌入因子  $\alpha$  与该方差成正比, 然后按照下式嵌入:

$$X_i^W = X_i + \alpha p_i \quad (6-2)$$

式中,  $x_i$  和  $x_i^w$  分别表示嵌入前后的左上角坐标为  $(u_i, v_i)$  和右下角坐标为  $(u_i + k - 1, v_i + k - 1)$  的图像块。由于嵌入过程要重复  $N$  次, 所以嵌入区域有可能互相重叠, 但这并不能影响水印的相关检测。

1	1	1	1	1	1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	1	-1	-1	1	-1	1	1	-1
1	1	1	1	1	1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	-1	1	1	-1	1	-1	-1	1
-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	1	1	1	1	-1	-1	-1	1	1	-1	1	-1	-1	1
-1	-1	-1	-1	1	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	1	1	1	1	-1	-1	1	-1	-1	1	-1	1	1	-1

图 6-3 8 种基本补丁形式 (以  $4 \times 4$  为例)

## (2) 利用黑白像素个数奇偶性的水印算法

这种方法用于二值图像，因为二值图像只有两个像素值，在其中嵌入水印信息就需要考虑其特殊性。这种嵌入方法是基于关系的，通过修改载体数据使得水印的不同取值反映了不同的关系，如大小关系、逻辑关系和奇偶性，以便于在检测时根据关系得到相应的水印信息。这种方法主要分为两步。

1) 首先是对一个图像进行  $3 \times 3$  邻域内中心像素的可翻转优先级的评估，通过中心像素周围的 8 个像素来给中心像素打分，分数越高的说明修改中心像素对视觉的影响越严重。而这个分数由平滑度和连通性决定。平滑度由水平、垂直和对角的像素由 0 变为 1 或者由 1 变为 0 的次数来评价。而连通性由黑像素聚类和白像素聚类的个数决定。图 6-4 给出的两种模式中，若评价中心像素由黑变白的效果，则左边所示模式比右边所示模式的分数低。算法实现时选择得分较低的所有像素构成可修改像素集合。

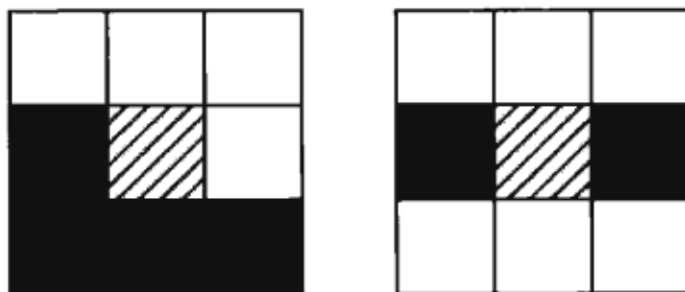



图 6-4 两个  $3 \times 3$  模式的中心像素的得分比较

2) 修改可修改像素集合中黑白像素个数的奇偶性来嵌入水印信息。如果要嵌入的水印信息为“0”，修改可以修改的像素，使得黑像素个数为偶数；如果要嵌入的水印信息为“1”，则修改可修改像素，使得黑像素的个数为奇数。

如果直接修改得分较低的像素，可能导致这些像素在修改之后变成不可修改像素，这样在检测过程中就需要原始图像的参与。

 通过增加块的大小可以增加安全性，如将块大小改为  $8 \times 8$ 。

## (3) 最低有效位 (LSB) 水印嵌入技术

最低有效位也称最不显著位，它是载体信息用二进制表示时的最低位。空域水印算法中最简单和最具有代表性的方案是用水印信息代替载体作品信息的最低有效位平面或者多个位平面，这时的水印信息通常是二值比特序列。

如果用 8 比特的二进制来表示灰度图像的每一个像素值，所有像素的最低位构成的位平面显现随机特性，而且改变最低位不会对视觉效果产生明显影响，因此，可以考虑用水印信息直接代替数字图像的最低位。其嵌入过程主要分为以下三步。

1) 将原始图像的空域像素值由十进制转换到二进制表示, 以  $3 \times 3$  大小的块图像为例, 如图 6-5 所示。

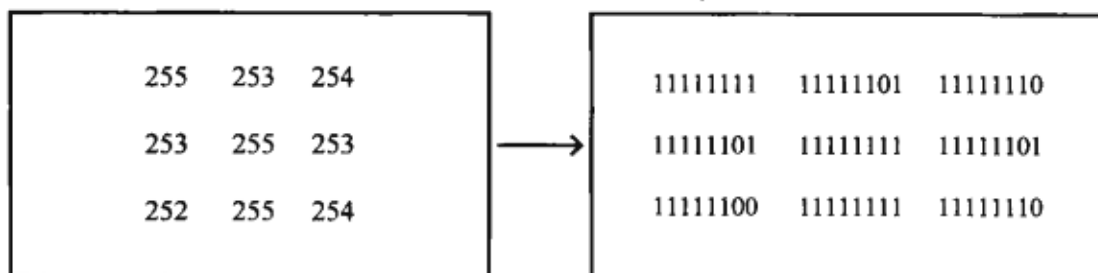


图 6-5 原始宿主图像的像素值用 8 比特的二进制表示

2) 用二进制水印信息中的每一比特信息替换与之相对应载体数据的最低有效位, 假设待嵌入的二进制水印信息序列为  $\{0, 1, 1, 0, 0, 0, 1, 0, 0\}$ , 则替换过程如图 6-6 所示。

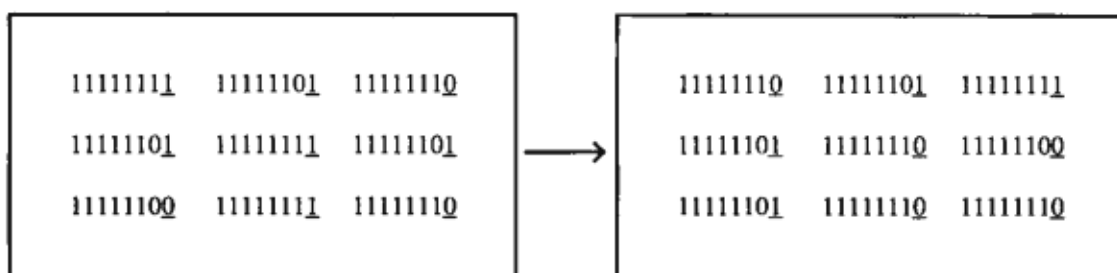


图 6-6 用二进制水印信息替换宿主数据的最低有效位

这个过程也可以用如下的嵌入公式来描述:

$$S_{i,j} = \begin{cases} X_{i,j} + W_{i,j} & (X_{i,j} \text{ 为偶数}) \\ X_{i,j} + W_{i,j} - 1 & (X_{i,j} \text{ 为奇数}) \end{cases} \quad (6-3)$$

其中,  $X_{i,j}$  表示第  $i$  行第  $j$  列的原始图像像素值,  $W_{i,j}$  为对应的待嵌入的二值水印。公式实际上是对载体图像像素的最低有效位清零, 然后在嵌入时直接加上二值水印。

3) 将得到的含水印的二进制数据转换为十进制像素值, 从而获得含水印的图像, 如图 6-7 所示。

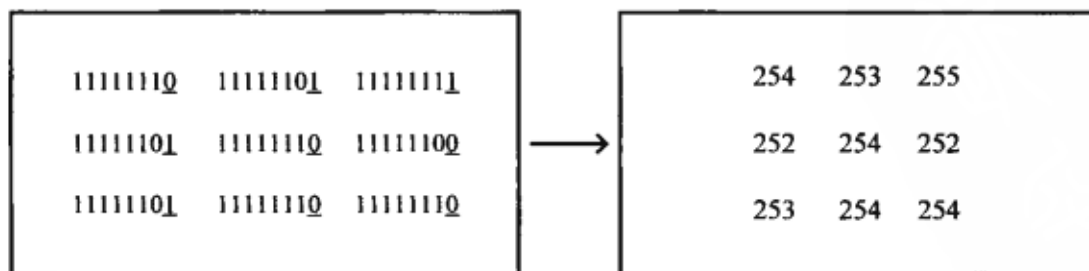


图 6-7 将替换之后的二进制数据转换为十进制像素值

水印的提取很简单, 只需将对应像素值转换为二进制形式, 然后提取最低有效位即可, 因此它可以实现盲检测。

#### (4) 不带嵌入因子的简单加性算法

这种加性规则不仅适用于时空域，还适用于变换域。通常在加性规则中都带有嵌入因子，当嵌入因子为 1 时，则表现为不带嵌入因子，嵌入因子的主要作用是调整嵌入水印的不可见性和鲁棒性。在时空域内，嵌入公式为

$$x^w = x + \alpha w \quad (6-4)$$

其中， $x^w = \{x_i^w, 0 \leq i < N\}$  为含水印载体， $x = \{x_i, 0 \leq i < N\}$  和  $w = \{w_i, 0 \leq i < N\}$  分别为原始载体和水印， $\alpha$  为嵌入因子。

### 2. 变换域图像水印嵌入技术

时空域的图像水印嵌入技术存在两个缺点。第一是嵌入数据的信息量受限制，不能过大。第二是鲁棒性较差，在对水印后的图像进行量化、压缩和滤波的情况下，水印后的图像会发生很大变化。为了解决这两个问题，可以采用 DCT 变换域（频域）的方法来实现图像水印嵌入。其主要思想是数字载体首先进行一种特定的正交变换，该变换可以针对整个载体或者是载体的每一个具体部分。嵌入空间是载体的某一个频带或者是某些频带，这些频带对应的变换系数遵循一定的规则被修改、替换或交换。通常载体的低频信息反映了载体的主要轮廓，不应有较大的失真，水印的嵌入将影响不可见性；而载体的高频信息是人类感知系统不敏感的信息，通常被压缩技术所剔除，故在该频带嵌入水印，水印的鲁棒性较差。

变换域水印嵌入算法的主要优点是：物理意义清晰、可充分利用人类的感知特性、不可见性和鲁棒性好以及与压缩标准兼容。变换域数字水印技术主要包含离散余弦变换域、离散小波变换域、离散傅立里变换域、离散分数傅里叶变换域、哈德码变换域、Fresnel 变换域、矢量变换域、KLT 变换域、Gabor 变换域和 Zernike 变换域等。其中 DCT 域、DWT 域和 DFT 域比较常见。下面进行详细介绍。

#### (1) DCT 域图像水印嵌入技术

离散余弦变换 (Discrete Cosine Transform) 简称 DCT。任何连续的实对称函数的傅里叶变换中只含有余弦项，因此余弦变换与傅里叶变换一样有明确的物理意义，DCT 变换避免了傅里叶变换中的复数运算，它是基于实数的正交变换。虽然在离散余弦变换中分为一维、二维、三维，但在大部分图像水印嵌入技术中，往往选用二维离散余弦变换。这是因为二维离散余弦变换不但能够将自然图像的主要信息集中到最少的低频系数上，而且引起的图像块效应最小，能够实现信息集中能力和计算复杂性的良好折中，它在压缩编码中得到广泛的应用。对于一副  $N \times N$  的图像  $s(x, y)$  来说，它的 DCT 变换为

$$S(u, v) = \frac{2}{N} c(u) c(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} s(x, y) \cos\left(\frac{\pi u(2x+1)}{2N}\right) \cos\left(\frac{\pi v(2y+1)}{2N}\right) \quad (6-5)$$



反离散余弦变换为

$$S(u, v) = \frac{2}{N} c(u) c(v) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} s(x, y) \cos\left(\frac{\pi u(2x+1)}{2N}\right) \cos\left(\frac{\pi v(2y+1)}{2N}\right) \quad (6-6)$$

特别地，在二维离散余弦变换基础上建立了数字图像的 JPEG 有损压缩标准。在该标准的基本模型中，数字图像首先被分割成  $8 \times 8$  的字块，然后经过基块离散余弦变换、系数量化和熵编码等过程最终实现了图像的有损压缩。基于 DCT 变换的水印嵌入技术主要包括以下几部分：

1) 将要嵌入的水印信息记为  $b(j)$ , ( $j=1, 2, \dots, 7$ )，其中  $b(j)$  为 0 或 1。将原始图像按  $8 \times 8$  像素分块，将水印信息周期性地扩展，即在  $8 \times 8$  像素分块中只包含 1bit 的水印信息，对每一块做 DCT 变换，得到如图 6-8 所示的 DCT 系数矩阵，其中每个格代表一个系数。有字母的格子代表嵌入水印信息的位置。

						$a(1,1)$	$a(1,2)$
					$a(2,1)$	$a(2,2)$	$a(1,3)$
				$a(3,1)$	$a(3,2)$	$a(2,2)$	
			$a(4,1)$	$a(4,2)$	$a(3,3)$		
		$a(5,1)$	$a(5,2)$	$a(4,3)$			
	$a(6,1)$	$a(6,2)$	$a(5,3)$				
$a(7,1)$	$a(7,2)$	$a(6,3)$					
	$a(7,3)$						

图 6-8 DCT 变换系数矩阵

2) 将中频系数进行分组，如图 6-8 所示，每 3 个系数一组，共分了 7 组， $b(j)$  代表该块要嵌入的第  $j$  比特位水印信息。由于在表 6-1 中只嵌入 1bit，而每一组的 3 个系数  $a(i,1), a(i,2), a(i,3)$  ( $i=1, 2, \dots, 7$ ) 都可以表达 1bit 的信息，因而实际上可以嵌入 7bit，由于需要将水印信息放大，在这 7 组中只嵌入 1bit，目的是进一步降低水印信息在提取时的错误，因而在这 7 组中嵌入的信息是一致的，即每一组中的序关系是一样的。

3) 水印信息是通过调整这 3 个数据的位置来实现的。当水印的第  $j$  比特位为 1 时，即  $b(j)=1$  时，将  $a(i,2)$  位置的系数与  $a(i,1), a(i,2), a(i,3)$  中绝对值最大的系数调换位置，若  $a(i,2)$  为绝对值最大的系数，则位置不变，否则将  $a(i,1), a(i,2), a(i,3)$  中绝对值最大的系数放在  $a(i,2)$  位置上，而  $a(i,1)$  的位置放绝对值次大的系数， $a(i,3)$  的位置放绝对值最小的系数；当水印的第  $j$  比特位为 0 时，即  $b(j)=0$  时，将  $a(i,2)$  位置的系数与  $a(i,1), a(i,2), a(i,3)$  中绝对值最小的系数调换位置，而  $a(i,1)$  的位置放绝对值最大的系数， $a(i,3)$  的位置放绝对值次大的系数。

因为嵌入水印时修改的 DCT 系数均为中频系数，系数值相对都比较接近，为增加水印的强度，需要选择适当的参数  $d$ ，通过参数  $d$  来调节嵌入水印的强度。但是，若参数  $d$  的选取过大，将对图像质量造成影响。

### (2) DWT 域图像水印嵌入技术

离散余弦变换是从图像空间到频率空间的全局变换,而离散小波变换(DWT)是一种局部的变换。由于离散余弦变换的全局本质,在变换空间中任何一个数据的误差都会影响到图像中的每一个像素,为了限制离散余弦变换的全局影响,JPEG 压缩标准把图像分成了一系列  $8 \times 8$  的小块。但是,这样一来在进行压缩时就不可避免地出现了“块效应”。此外,离散小波变换的另一个特点是它具有多尺度分析的能力。因此,当前最新的图像压缩标准——JPEG2000 和视频的 MPEG7 压缩标准都采用了小波变换。利用小波变换把原始图像分解成多频段的图像,能适应人眼的视觉特性且使得水印的嵌入和检测可分为多个层次进行,小波变换域数字水印方法兼具时空域方法和 DCT 变换域方法的特点。因此,基于离散小波变换的数字水印算法已经成为当前研究的热点和非常重要的研究方向。

在小波变换的图像水印算法中有一种多分辨率分解的方法很好推广,该算法利用灰度级二维数字水印,不仅可实现多重水印的嵌入,而且灰度级数字水印图像所包含的信息量、可感知性、可辨别性及保密性是传统二值水印和伪随机序列水印所无法比拟的。利用图像的多分辨率分解技术,相同分辨率层次的数字水印嵌入到对应的相同分辨率层次的原始静态图像之中,使水印对原始图像具有自适性。由于水印的嵌入过程是基于原始图像的不同分辨率层次之间的关系,所以水印的提取过程不需要原始图像。在水印嵌入的过程中,需要将原始图像通过二维小波变换分解为三层多分辨率金字塔结构,如图 6-9 所示。其中,多分辨率分解的第三层中,最低频子带  $LL_3$  包含了原始图像的最低分辨率信息,而  $HL_3$ 、 $LH_3$  和  $HH_3$  是  $LL_3$  的精细图像信息,第三层中  $HL_3$ 、 $LH_3$  和  $HH_3$  图像包含了第二层参考图像( $HL_2$ 、 $LH_2$  和  $HH_2$ )的粗糙信息。在以小波方式分解的图像数据中可以构成若干如图 6-9 所示的四叉树。在每个四叉树中,粗糙信息层中的小波系数是其下一个精细层次中的四个对应位置的小波系数的父节点,它代表了精细层中的四个对应位置的小波系数的幅度平均值,而这四个对应位置的小波系数成为其父节点的子节点。

由于分解后的三层水印  $G_2$ 、 $L_1$  和  $L_0$  仍然是灰度级图像,为了方便地实现水印的嵌入,利用位分解方法,把  $G_2$ 、 $L_1$  和  $L_0$  进一步分解为一系列二值位平面的形式,以作为镶嵌水印的掩码信息。对于一个具有 256 级灰度(即 8 位灰度级)的图像,利用灰度级图像的位分解方法,可以将其分解为 8 个二值位平面。这些位平面就可以作为水印嵌入过程中的掩码信息。如果把每个位平面作为一个水印信息,那么 8 个位平面就可以实现多重水印的镶嵌。

### (3) DFT 域图像水印嵌入技术

离散傅里叶变换(Discrete Fourier Transform, DFT)是线性系统分析的有力工具,在数字信号处理技术中占有重要的地位。由于离散傅里叶变换是正交变换,计算时可以采用快速算法,特别地,信号的离散傅里叶变换系数有明确的物理意义,因此它在通信、雷达、声纳、遥感、医学、图像处理、语音合成与分析等许多领域得到了广泛的应用。离散傅里叶变换在数字水印处理技术

中也受到了高度的重视。离散傅里叶变换是复数变换，在幅度和相位满足特定的条件时，数字水印信息既可以嵌入到媒体信号的幅度上，也可以隐藏在它的相位中。下面主要介绍傅里叶变换在二维中的定义以及嵌入条件。

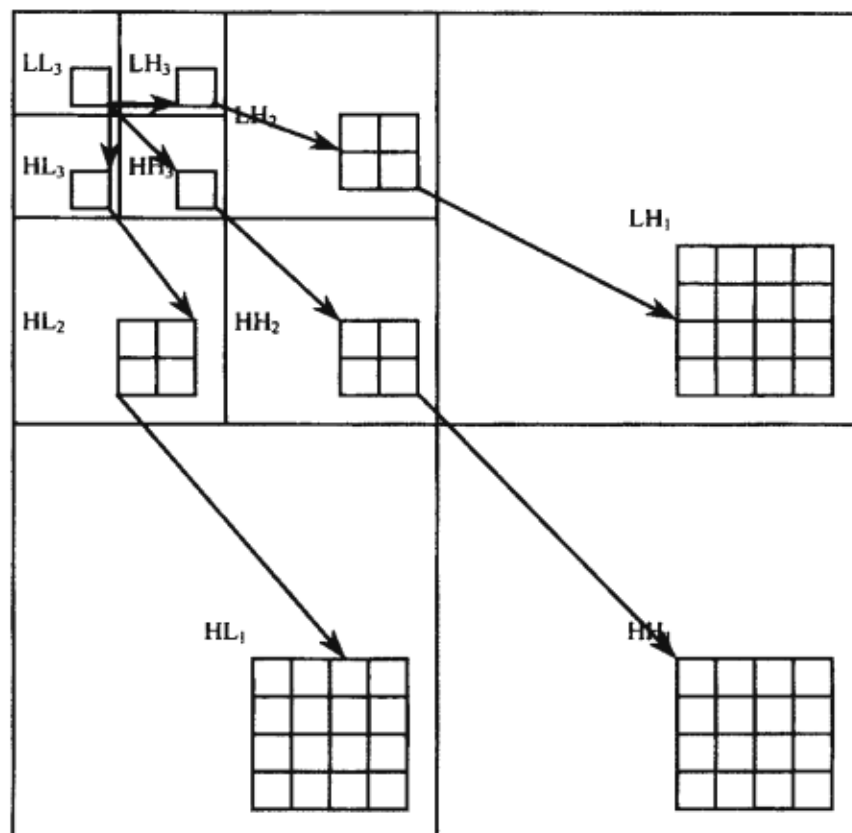


图 6-9 基于小波变换的原始图像的多分辨率分解

二维离散傅里叶变换（2D-DFT）及其逆变换（2D-IDFT）分别定位为

$$X_{uv} = \sum_{i=0}^{N_1-1} \sum_{k=0}^{N_2-1} x_{ik} e^{-j\left(\frac{2\pi}{N_1}iu + \frac{2\pi}{N_2}kv\right)} \quad (6-7)$$

$$x_{ik} = \frac{1}{N_1 N_2} \sum_{u=0}^{N_1-1} \sum_{v=0}^{N_2-1} x_{uv} e^{j\left(\frac{2\pi}{N_1}iu + \frac{2\pi}{N_2}kv\right)} \quad (6-8)$$

如果  $x_{ik}$  是实数（如数字图像的采样数据），则有

$$X_{uv} = X_{(N_1-u)(N_2-v)}^* \quad (6-9)$$

可以通过适当修改 DFT 系数的幅度  $|X_{uv}|$  嵌入水印。为保证修改结果的离散傅里叶逆变换为实数，修改幅度  $|X_{uv}|$  应满足正对称条件，即

$$|X_{uv}| \leftarrow |X_{uv}| + \varepsilon \quad (6-10)$$

$$|X_{(N_1-u)(N_2-v)}| \leftarrow |X_{(N_1-u)(N_2-v)}| + \varepsilon \quad (6-11)$$

其中  $\varepsilon$  为幅度修改量。

同样,也可以通过适当修改 DFT 系数的相位  $\angle X_{uv}$  嵌入水印。为了保证修改结果的离散傅里叶逆变换也为实数,修改相位  $\angle X_{uv}$  应满足负对称条件,即

$$\angle X_{uv} \leftarrow \angle X_{uv} + \delta \quad (6-12)$$

$$\angle X_{(N_1-u)(N_2-v)} \leftarrow \angle X_{(N_1-u)(N_2-v)} - \delta \quad (6-13)$$

其中  $\delta$  为相位修改量。

考虑到傅里叶变换系数幅度的平移不变性,许多文献提出通过修改幅度达到鲁棒嵌入的目的,这里介绍一下基于幅度调制的嵌入方法。基于幅度调制的嵌入方法不外乎加性和乘性两种。为了使修改后系数的逆变换仍是实数,修改幅度要满足正对称的条件。由此,加性幅度调制可以表示为

$$|X_{i+M_1}^w| = |X_{i+M_1}| + \alpha \omega_i \quad (6-14)$$

$$|X_{N-i-M_1}^w| = |X_{N-i-M_1}| + \alpha \omega_i \quad (6-15)$$

而乘性幅度调制可表示为

$$|X_{i+M_1}^w| = |X_{i+M_1}|(1 + \alpha \omega_i) \quad (6-16)$$

$$|X_{N-i-M_1}^w| = |X_{N-i-M_1}|(1 + \alpha \omega_i) \quad (6-17)$$

式中,  $0 \leq i < M, M = M_2 - M_1 + 1, 0 < M_1 < M_2 < \frac{N}{2}$ ,  $\alpha$  为嵌入强度。

### 6.1.2 图像水印提取技术

数字水印的提取算法是数字水印系统的关键部分之一。所谓水印提取,是指根据提取密钥通过一定的算法提取出可疑作品中的每个印记,其长度等于原始水印序列的长度。如果水印提取过程中需要用到原始载体,则称此过程为明提取。相反,如果水印提取过程不需要用到原始载体,则称此过程为盲提取。因此,水印提取过程往往与水印嵌入算法密切相关,在水印提取之前通常要先进行水印检测。因为水印的提取通常伴随着水印的嵌入,所以这里仅阐述目前比较常用的一些嵌入方法(如公钥水印嵌入算法、DCT 水印嵌入算法和基于小波的半脆弱水印嵌入算法)其相应的提取方法。

#### 1. 公钥水印提取技术

公钥水印中算法包含了 Torus 自同构映射、公钥密码 RSA 和哈达玛变换等多种方法,这里主要通过哈达玛变换来介绍一下公钥水印的一种提取方法。哈达玛嵌入水印的思路大致分为进行水印置换,水印数据混合编码,图像块变换,修改 DCT 系数和逆 DCT 变换五步,而其水印提取的过程并不是单纯的一次逆向提取。这里是由于公钥水印方案中是有两个水印:IPR 发布的公开水印和创作者提取水印需要一些条件,测试图像和一些参数。比如要提取公开水印则需要知道水印

公钥即置乱变换的参数,如果要提取私有水印则需要水印私钥。所以其提取水印的具体过程如下。

1) 块变换。测试图像分为  $M \times M$  个  $8 \times 8$  的块,对每块进行 DCT 变换  $DI_{m,n} = DCT(BTI_{m,n})$ 。

2) 提取 2 个水印的混合编码 EHW, 具体过程如图 6-10 所示。

3) 分解混合编码数据。根据哈达玛矩阵的性质,可以从混合编码数据中根据所要检测的水印得到打乱后的水印数据: EPW1 和 EPW2。

4) 恢复水印。先将由 EPW1 或 EPW2 得到的  $\{BEPW'_{m,n}(i,j), 1 \leq i, j \leq 2\}$  重组得到置乱的水印 EW1 或 EW2。然后,根据检测者知道的水印公钥或私钥,由 Torus 自同构变换可重复特性恢复提取出的水印 EW (EW1 或 EW2):  $EW(k,1) = EPW(i,j)$ 。其中,  $(k,1)$  与  $(i,j)$  之间满足 Torus 自同构变换的关系。 $(k,1)$  为变换前的像素坐标,  $(i,j)$  为变换后的该像素的坐标。

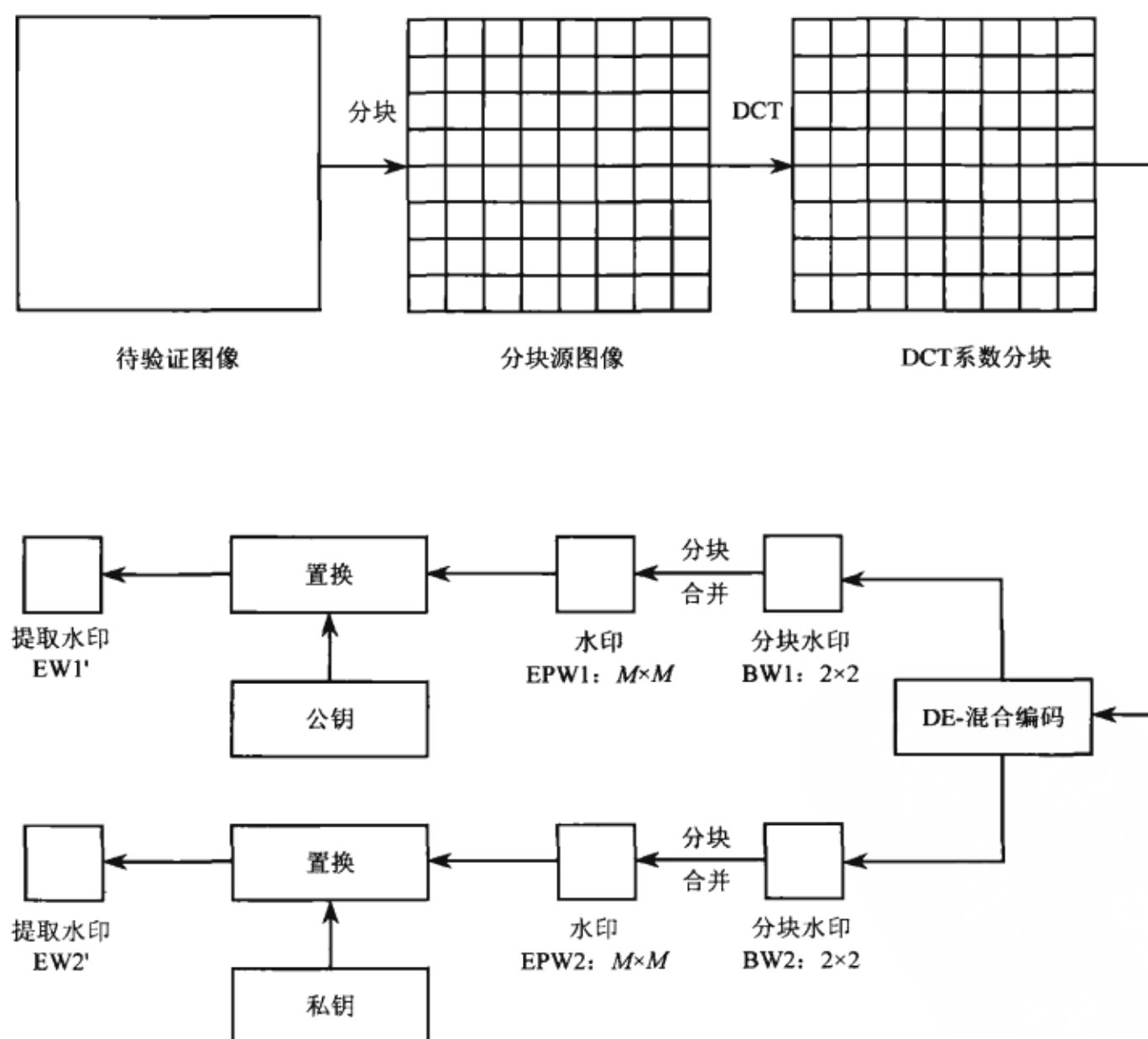


图 6-10 公钥水印提取算法

## 2. 基于 DCT 的水印提取技术

这里可以与上文所提到的 DCT 方式嵌入水印技术相结合来看。现就 DCT 中基于系数比较算法的水印提取技术来阐述，这种水印提取的方法需要满足两个最大隶属原则两次。水印的提取算法也就是水印嵌入算法的逆向算法，但是由于数字图像在打印/扫描后，失真较大，“放大”了的水印信息在按块提取时，提取出的数据往往是不一致的。为此考虑用模糊模式识别解决。

对经过打印/扫描后的图像按  $8 \times 8$  进行分块，对于图像的每一个  $8 \times 8$  块，做 DCT 变换，所得系数仍用图 6-8 来表示。根据图像的第  $j$  个  $8 \times 8$  分块来决定隐藏数据的第  $j$  个比特位  $b(j)$  的值。

具体提取算法如下：

```
if |a(i,2)| >= (|a(i,1)| + |a(i,2)|) / 2
    b(i,j)=1;
else
    b(i,j)=0;
```

由于存在误判问题，由此提取的 7 个比特  $b(1,j), b(2,j), \dots, b(7,j)$  可能是不一致的，通过提取出的  $b(1,j), b(2,j), \dots, b(7,j)$ ，定义  $b(j)$  对 1 的隶属度为  $b(1,j), b(2,j), \dots, b(7,j)$  中 1 的个数除以 7。对 0 的隶属度为  $b(1,j), b(2,j), \dots, b(7,j)$  中 0 的个数除以 7。根据模糊模式识别的最大隶属度原则，来确定  $b(j)$  是 1 还是 0。

由于图像经过打印/扫描后失真较大，并且图像的边缘比起内部来，误提取的概率要大，因此需要将水印信息进行重复嵌入。在相同位提取时，再次使用模糊模式识别的最大隶属度原则来解决提取出的数据不一致的问题，若记组成水印信息的 0-1 序列的长度是  $N$ ，重复嵌入的次数为  $K$ ，提取出的  $K$  个序列记为

$$W_1 = (w_1^{(1)}, w_2^{(1)}, \dots, w_i^{(1)}, \dots, w_N^{(1)}); \quad (6-18)$$

$$W_2 = (w_1^{(2)}, w_2^{(2)}, \dots, w_i^{(2)}, \dots, w_N^{(2)}); \quad (6-19)$$

⋮

$$W_k = (w_1^{(K)}, w_2^{(K)}, \dots, w_i^{(K)}, \dots, w_N^{(K)}); \quad (6-20)$$

将提取的水印序列记为  $W = (w_1, w_2, \dots, w_i, \dots, w_N)$ ，对于它的第  $i$  个比特位  $w_i = (1, 2, \dots, N)$ ，定义  $w_i$  对 1 的隶属度为  $w_i^{(1)}, w_i^{(2)}, \dots, w_i^{(K)}$  中 1 的个数除以  $K$ 。对 0 的隶属度为  $w_i^{(1)}, w_i^{(2)}, \dots, w_i^{(K)}$  中 0 的个数除以  $K$ 。根据模糊模式识别的最大隶属度原则来确定  $w_i$  是 1 还是 0。

## 3. 基于小波域的半脆弱型水印算法提取技术

由于小波变换是一种空间到频率的分析方法，能同时反映图像的空间位置和频率。小波变换的局部化作用能够检测到图像被篡改的区域，而小波变化的频率域则反映了被篡改的尺度，能够满足图像内容级认证的要求，而且静止图像压缩标准 JPEG2000 也是基于小波变换的，故基于小波域的数字水印认证技术是当前的研究热点之一。



这里介绍一种基于离散小波变换的用于彩色图像的内容级认证水印算法的提取。该算法主要思想是首先将 RGB 真彩色图像转换为 YUV 彩色图像,将原始图像 Y 分量进行 4 级小波分解,把每一层的 3 个细节分量分块,采用均值调制的方法嵌入水印,并且通过随机生成个人的密钥来防止水印的假冒。该算法的优点是提供了可以显示图像修改程度和修改区域的信息,这些信息用来估计图像改变的特性。

图 6-11 说明了利用这种方法提取数字水印的过程。这个过程就是要把嵌入在特定位置的数字水印提取出来,是嵌入过程的逆过程。首先对可能经受过某种程度改变的含水印图像进行 L 级离散小波变换,得到各层的系数矩阵,对各层系数矩阵按  $(3 \times 3, 4 \times 4, 4 \times 4, 4 \times 3)$  进行分块,使用密钥  $K_c$  确定水印嵌入位置,提取出对应块的均值量化值。然后,对恢复出的数字水印与原来的数字水印的变化进行比较,以此来判断图像的变化情况。

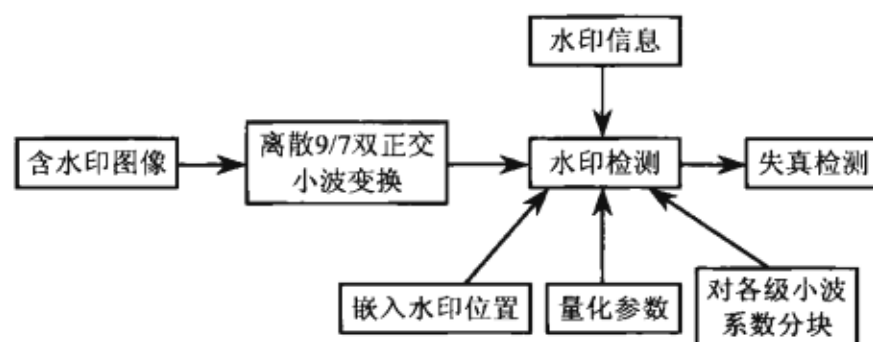


图 6-11 半脆弱水印的提取过程

使用函数  $T(x)$  来表示小波系数  $x$  所对应的水印是否发生变化,即

$$T(x) = \begin{cases} 1, & \text{if } Q(x, q) \neq w \oplus k_q \\ 0, & \text{if } Q(x, q) = w \oplus k_q \end{cases} \quad (6-21)$$

根据水印篡改位置的 8 个邻域是否仍为被篡改的区域,将篡改区域分为密集区和稀疏区。由于被恶意篡改的图像对应的水印变化大多比较集中,而 JPEG 压缩等一般性操作对整幅图像处理,产生的水印变化比较分散,因此可以通过篡改区域的密集程度区分恶意的攻击和一般的图像处理。

用  $N_l^t$ 、 $N_l^p$ 、 $N_l^d$  和  $N_l^s$  分别表示  $l$  级的分块总数、篡改系数总块数、篡改系数密集总数和篡改系数稀疏总数,其中  $N_l^p = N_l^d + N_l^s$ 。定义  $l$  的篡改率为  $TR_l = N_l^p / N_l^t$ 。 $TR_l$  用来量度篡改程度。同时使用以下规则来判定图像是否经历了修改以及修改是否为恶意的。

- 规则 1: 若每一级所对应的  $TR_l = 0$ , 那么图像没经过任何修改。
- 规则 2: 若每一级所对应的  $TR_l = 0$ , 那么图像仅经正常的图像处理。
- 规则 3: 假定  $l^*$  满足关系式  $TR_{l^*} = \min_l (TR_l)$  的小波分解级, 若  $TR_{l^*} > 0$  且  $N_{l^*}^d < \alpha \times N_{l^*}^p$  ( $0.5 \leq \alpha \leq 1.0$ ), 则仅经正常图像处理。
- 规则 4: 若每一级都有  $N_l^p = N_l^d$ , 那么图像仅经过恶意篡改。

- 规则 5: 若以上规则均不符合, 那么图像同时被恶意和偶然篡改。

对于某一图像, 如果满足前 3 条规则之一的话, 就认为图像未被篡改, 仍然可信, 否则就认为图像已经篡改。

## 6.2 系统功能

数字图像水印系统的主要功能是根据数字图像水印的基本模型, 从实际数字图像水印处理的需求中衍生出来的, 其主要包括四个步骤, 水印的生成、嵌入、检测和提取。而整个水印系统还应包括外界的攻击过程。而数字图像水印系统的主要需求是从鲁棒性、透明性、安全性、数据容量、盲检测和自恢复性, 以及确定性这 6 个方面来说的。下面具体介绍一下数字图像水印系统的功能。

### 6.2.1 功能描述

数字图像水印系统包含两种水印嵌入方法, 为了方便学习, 从不同的嵌入算法分别介绍两种水印方法的功能。第一种是数字图像水印系统中采用不带嵌入因子的加性规则算法, 第二种是采用基于位平面的最低有效位 (Least Significant Bit, LSB) 替换算法实现水印的嵌入。

#### 1. 不带嵌入因子的加性规则

数字图像水印系统采用的第一种嵌入方式为不带嵌入因子的加性规则方法进行嵌入, 该部分系统的功能描述如下。

- 1) 载入并显示宿主图像。通过应用程序将宿主数字图片载入到程序之中并予以图片显示。
- 2) 载入宿主图像判别。对载入数字图片进行大小、格式、位的提取并进行判断, 是否满足简单加性规则算法的基本要求。
- 3) 载入并显示水印图像。对载入水印图片进行大小、格式、位的提取并进行判断, 是否满足嵌入要求, 要实现与宿主图像之间的比较判断, 判断通过后在嵌入水印图像位置将图像显示出来。
- 4) 嵌入水印。通过简单加性规则将预先载入的水印图片嵌入到宿主图片之中, 并在程序中用扫描的形式显示出嵌入后的图片。
- 5) 保存水印过的图像。将载入水印后的图像通过程序保存到本地计算机硬盘中。
- 6) 打开水印过的图像。将保存在本地计算机硬盘中的水印过的图像加载进程序之中并予以水印检测。
- 7) 提取水印。对载入程序中的已嵌入水印图像进行水印提取操作, 这里要对水印的大小格式进行判断。
- 6) 按钮可用控制。每一步都与上一步有一定的逻辑关系的, 只有完成前一步所需操作, 下一步按钮才可以使用。

## 2. 最低有效位替换

数字图像水印系统采用的第二种嵌入方式为最低有效位替换,该部分系统的功能描述如下。

1) 载入并显示宿主图像。通过应用程序将宿主数字图片载入到程序之中并予以图片显示。

2) 载入宿主图像判别。对载入数字图片进行大小、格式、位的提取并进行判断,判断是否满足 24 位 BMP 位图的基本要求。

3) 载入并显示水印图像。对载入水印图片进行大小、格式、位的提取,并进行判断是否满足嵌入要求,要实现与宿主图像之间的比较判断,判断通过后在嵌入水印图像位置将图像显示出来。

4) 嵌入水印。通过 LSB 最低有效位替换将预先载入的水印图片嵌入到宿主图片之中。

5) 保存水印过的图像。将载入水印后的图像通过程序保存到本地计算机硬盘中。

6) 打开水印过的图像。将保存在本地计算机硬盘中的水印过的图像加载进程序之中并予以水印检测。

7) 提取水印。对载入程序中的已嵌入水印图像进行水印提取操作,这里要对水印的大小格式进行判断。

8) 按钮可用控制。每一步都与上一步有一定的逻辑关系的,只有完成前一步所需操作,下一步按钮才可以使用。

## 6.2.2 界面效果

数字图像水印系统的界面效果如图 6-12 所示,图中显示的是水印嵌入和提取的效果,窗口中左上角的图像为宿主图像,右上角的图像为水印图像,左下角的图像为嵌入水印的图像,右下角的图像为提取的水印图像。

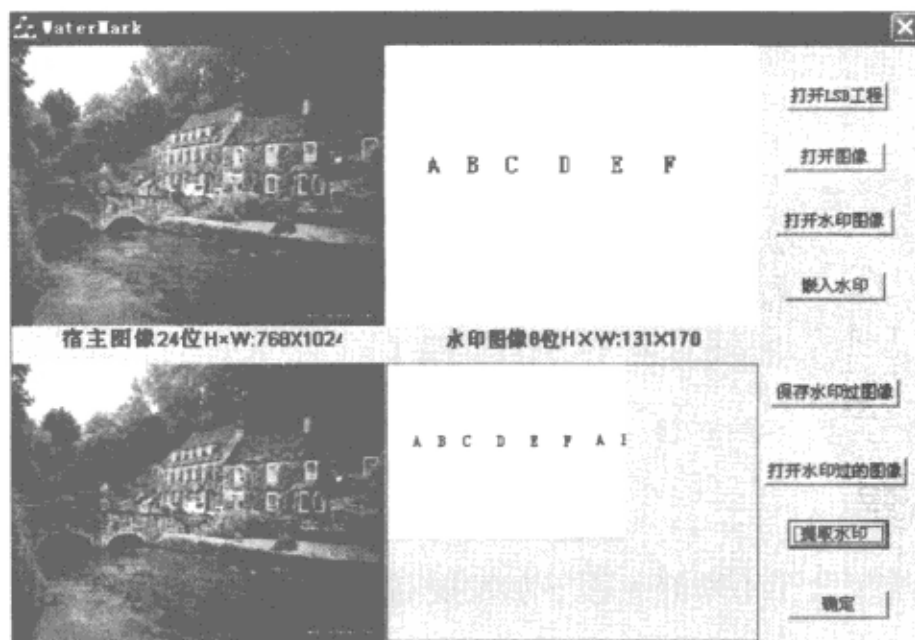


图 6-12 数字水印系统界面效果

## 6.3 系统结构与流程

数字图像水印系统主要包含了水印的生成、嵌入、检测和提取4个主要部分。下面具体介绍数字图像水印的系统结构与主要流程。

### 6.3.1 总体结构

数字图像水印系统通常包括嵌入器和检测器两部分。嵌入器至少具有两个输入量，一个是原始图像信息，即宿主图像。另一个就是水印图像，它通过适当变换之后作为待嵌入的水印信号。图6-13给出了数字图像水印系统总体结构，该结构主要包括了简单加性规则和最低有效位两种算法的水印生成、嵌入、检测和提取部分。

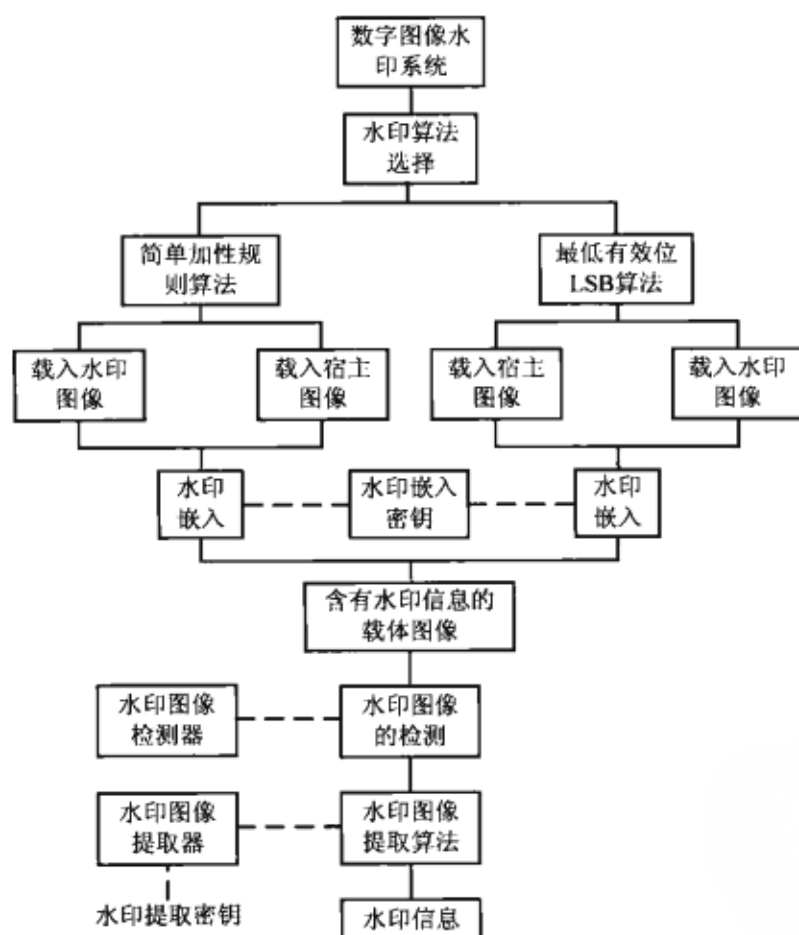


图 6-13 数字图像水印系统总体结构图

### 6.3.2 主要流程

数字图像水印系统包含了不带嵌入因子加性规则算法的嵌入提取以及 LSB 最低有效位算法的嵌入与提取。

## 数字图像处理典型案例详解

### 1. 不带嵌入因子的加性规则算法主要流程

数字图像水印系统的不带嵌入因子加性规则算法的嵌入流程主要是实现水印信息加密后秘密隐藏于宿主图像的功能, 根据这个特点, 图 6-14 给出了不带嵌入因子加性规则算法的数字图像水印系统的嵌入流程。

数字图像水印系统不带嵌入因子的加性规则算法的提取部分主要实现将水印信息从宿主图像中提取出来的功能。图 6-15 给出了数字图像水印系统中不带嵌入因子的加性规则算法的提取流程。

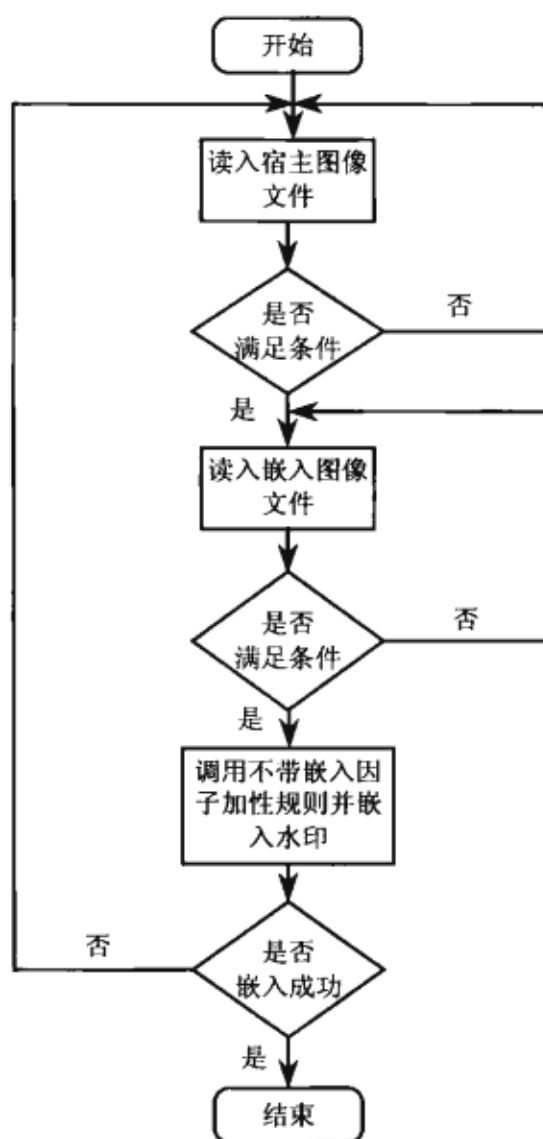


图 6-14 数字图像水印系统加性规则算法嵌入水印流程图

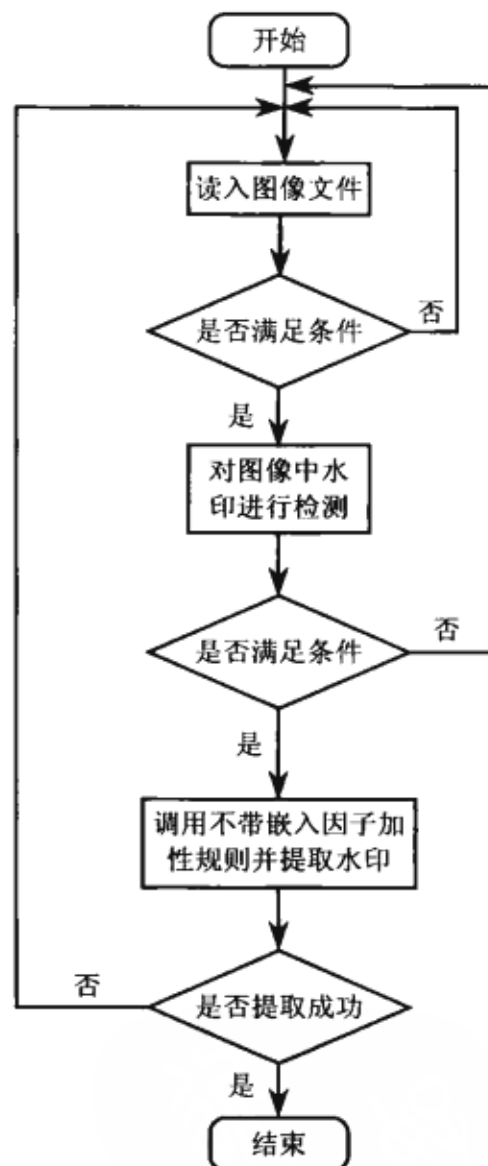


图 6-15 数字图像水印系统加性规则算法提取水印流程图

### 2. 最低有效位 (LSB) 算法主要流程

数字图像水印系统的最低有效位算法的嵌入流程主要是采用 LSB 算法对水印信息进行加密, 将彩色水印图像隐藏于宿主图像之中, 根据这个特点, 图 6-16 给出了最低有效位算法的数字图像水印系统的嵌入流程。在水印的嵌入这个环节中, 隐藏信息的生成与水印的嵌入被放入同一环节。

数字图像水印系统最低有效位算法的提取部分主要实现将水印信息从宿主图像中提取出来的功能。图 6-17 给出了数字图像水印系统中最低有效位算法的提取流程。其中在水印的提取中包含了水印的检测。

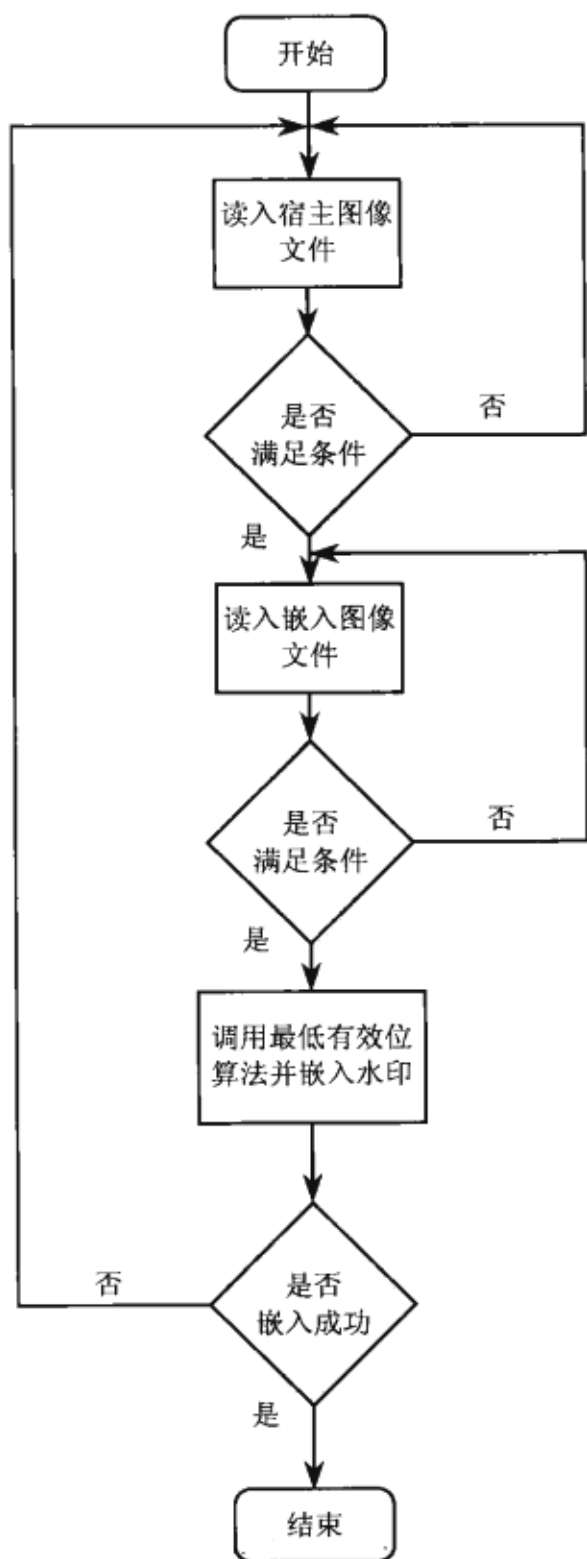


图 6-16 数字图像水印系统最低有效位算法嵌入水印流程图

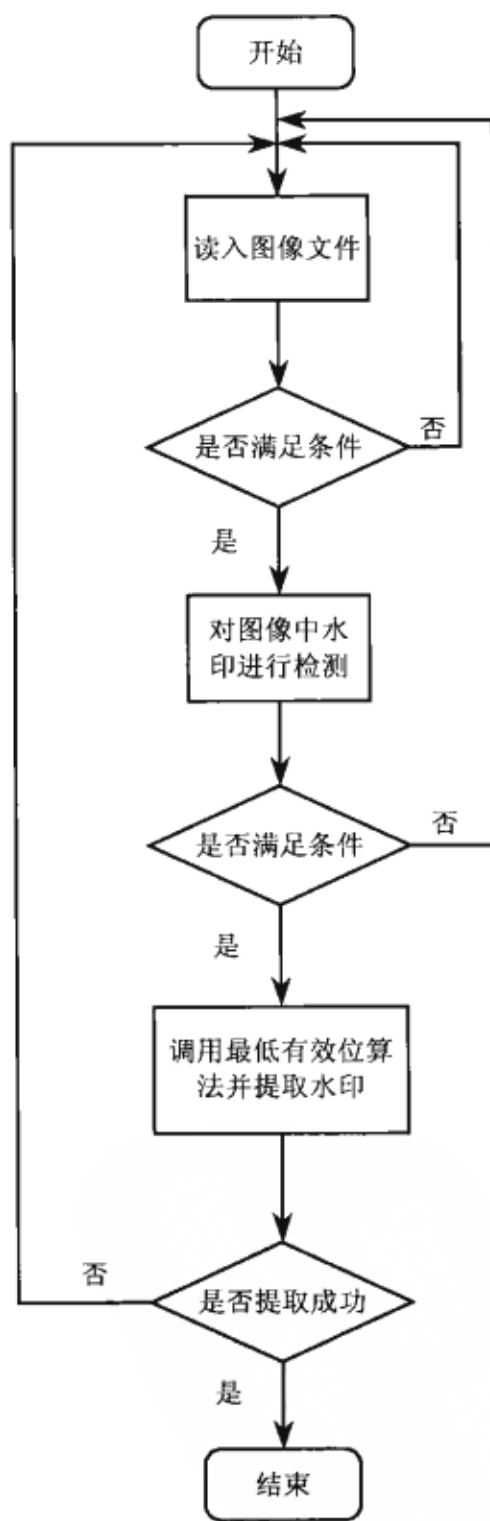


图 6-17 数字图像水印系统最低有效位算法提取水印流程图



### 3. 数字图像水印系统总流程图

数字图像水印系统包含了两种算法的水印生成、嵌入、检测和提取。系统总流程图如图 6-18 所示。

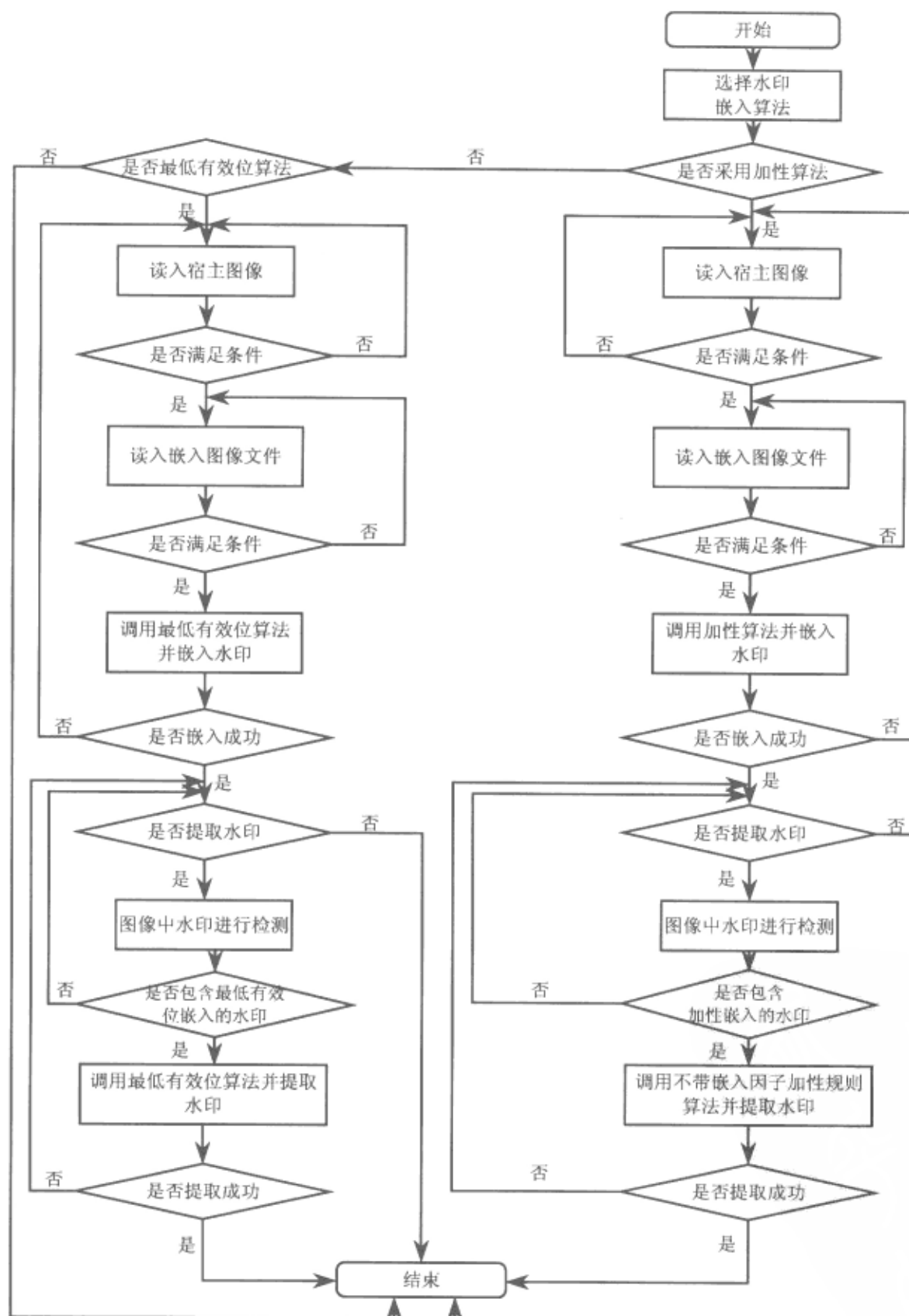


图 6-18 数字图像水印系统总流程图

## 6.4 编程实现

数字图像水印系统采用 VC 2008 开发平台编程实现。

### 6.4.1 不带嵌入因子的加性规则算法实现

首先是程序中打开宿主图像的函数 OnOpenFile(), 这部分主要实现将数字图像打开并对图像进行验证, 判断其是否符合标准, 该函数的函数体如下:

```
void CWaterMarkDlg::OnOpenFile()
{
    CFileDialog dlg(TRUE, "", NULL,
        OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT | OFN_ALLOWMULTISELECT,
        "(*.bmp)|*.bmp|所有文件 (*.*)|*.*||", AfxGetMainWnd());
    //读取图像的文件名 CString filename;
    if (dlg.DoModal() == IDOK)
    {
        //Bitmap2Data() 将图像转换为数据保存在二维数组 m_tOriPixelArray 中
        POSITION pos = dlg.GetStartPosition();
        filename = dlg.GetNextPathName(pos);
        Bitmap2Data();
        if (hwnd[0] != NULL)
            hwnd[0] = NULL;
        if (hSrcDC[0] != NULL)
            hSrcDC[0] = NULL;
        if (hDesDC[0] != NULL)
            hDesDC[0] = NULL;
        hwnd[0] = GetDlgItem(IDC_STATIC);
        hDesDC[0] = hwnd[0] -> GetDC() -> m_hDC;
        hSrcDC[0] = CreateCompatibleDC(hDesDC[0]);

        filename = dlg.GetPathName();
        hBitmap[0] = (HBITMAP) LoadImage(AfxGetInstanceHandle(), filename, IMAGE_BITMAP,
            0, 0, LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        GetObject(hBitmap[0], sizeof BITMAP, &bm[0]);
        SelectObject(hSrcDC[0], hBitmap[0]);
        hwnd[0] -> GetClientRect(&rect[0]);

        ::SetStretchBltMode(hDesDC[0], COLORONCOLOR);
        ::StretchBlt(hDesDC[0], rect[0].left, rect[0].top, rect[0].right, rect[0].
            bottom, hSrcDC[0], 0, 0, bm[0].bmWidth, bm[0].bmHeight, +SRCCOPY);
        show[0] = TRUE;
        SetTimer(NULL, 50, 0);
        GetDlgItem(IDC_BUTTON2) -> EnableWindow(TRUE);
    }
}
```

```

else
{
    TRACE("错误");
}
CString s,s0,s1,s2;
s.Format("%2d",bm[0].bmHeight);           //将宿主图像的高转化为二进制
s1.Format("%2d",bm[0].bmWidth);           //将宿主图像的宽转化为二进制
s0.Format("%d",m_dib->GetBiBitCount());    //将宿主图像的计数转化为十进制
s2="宿主图象"+s0+"位 H×W:"+s+"X"+s1;

if(pEdit!=NULL)
    pEdit=NULL;
r.left=35;r.top=rect[0].bottom+2;r.right=200+r.left;r.bottom=r.top+20;
pEdit=new CEdit;
pEdit->Create(ES_CENTER|WS_VISIBLE|ES_READONLY,r,this,1);
CFont * cFont=new CFont;
cFont->CreateFont(16,0,0,0,FW_SEMIBOLD,FALSE,FALSE,0, ANSI_CHARSET,OUT_
DEFAULT_PRECIS,CLIP_DEFAULT_PRECIS,DEFAULT_QUALITY,DEFAULT_PITCH&FF_SWISS,"Arial");
pEdit->SetFont(cFont,TRUE);
pEdit->SetWindowText(s2);
}

```

在此函数中，CString 变量中存放的是数字图像名称，对其格式进行判断，之后调用 Bitmap2Data()函数，该函数的作用是将 24 位的 BMP 图像通过逐行扫描存入到一个二维数组 m\_tOriPixelArray 中。接下来是对图像的读入信息进行处理，其中大部分属于 MFC 类库中的方法，这里不做一一阐述，获取图像基本信息后分别将宿主图像的高和宽的二进制、位数计数值的十进制表示和三者连在一起的表示保存在 s、s0、s1、s2 中。

这里介绍一下 Bitmap2Data()函数，其主要功能是将宿主图像转换为数据保存在二维数组中，该函数函数体如下：

```

void CWaterMarkDlg::Bitmap2Data()//将宿主图像转换为数据保存在二维数组 m_tOriPixelArray 中
{
    int i,j;
    if(m_tOriPixelArray !=NULL)           //判断数组是否为空并将其清空
        if(m_Ih!=0)
            for(i=0; i<m_Ih; i++)
                delete [] m_tOriPixelArray[i];
        else
            for(i=0; i<ImageHeight; i++)
                delete [] m_tOriPixelArray[i];
    if(filename.GetLength()==0)
    {
        MessageBox("无效的文件");
        return ;
    }
}


```

```

m_dib->Open(filename);           //打开宿主图像并对其进行转换
ImageWidth = m_dib->GetWidth();   //获取宿主图像的宽
ImageHeight = m_dib->GetHeight(); //获取宿主图像的高
biBitCount = m_dib->GetBiBitCount(); //获取宿主图像的计数值
BYTE *colorTable;
colorTable = (BYTE *) m_dib->m_pDibBits; //获得宿主图像的彩色表
int byteBitCount = m_dib->GetBiBitCount()/8;
m_tOriPixelArray = new RGBQUAD*[ImageHeight]; //定义二维数组大小
for (int l=0;l<ImageHeight;l++)
m_tOriPixelArray[l]=new RGBQUAD [ImageWidth];
int count = 0;
int num = 0;
    //分别对二维数组中的每个元素中的 RGB 值进行赋值操作
for(i=ImageHeight-1; i>=0; i--)
{
    for(j=0; j<ImageWidth; j++)
    {
        m_tOriPixelArray[i][j].rgbBlue =colorTable[count++];
        m_tOriPixelArray[i][j].rgbGreen=colorTable[count++];
        m_tOriPixelArray[i][j].rgbRed =colorTable[count++];
        m_tOriPixelArray[i][j].rgbReserved = 0;
        count += byteBitCount-3;
        num+=1;
    }
    count += (4-(ImageWidth*byteBitCount)%4)%4;
}
show[2]=TRUE;
}

```

函数首先是对保存宿主图像的变量 `m_lh` 等值进行判断, 如果不为空则将其值置 0, 之后从读入的宿主图像中获取其高、宽、比特位以及彩色表的指针地址。之后开始对宿主图片进行逐行逐列的扫描, 将其中每一个像素点的 RGB 分量值一一存放在 `m_tOriPixelArray` 二维数组中。

 `m_tOriPixelArray` 二维数组的类型为 `RGBQUAD`, 其中包含了 4 个分量的值, 分别是 `rgbBlue`、`rgbGreen`、`rgbRed` 和 `rgbReserved`。关于这种类型的具体信息请参考 MFC 中的 API 帮助文档, 这里不做详细解释, 读者只需理解二维数组中存放的是宿主图像每个像素点的 RGB 值即可。

同样, 在运行 `OnOpenWaterMark()` 函数对水印图像进行载入时, 也需要进行格式和大小的判断并将数字图像转换为数组数据, 此外, 这里会对水印图像与宿主图像的大小进行一次比较, 具体比较方法是水印图像宽和高的乘积的 2 倍要小于宿主图像宽和高的乘积。此外, 水印图像也同样需要调用一个函数 `WaterBitmap2Data()` 将水印数字图像转换到一个二维数组 `m_watermarkdata` 中, `OnOpenWaterMark()` 函数的函数体如下:

```

void CWaterMarkDlg::OnOpenWaterMark()
{
    CFileDialog dlg1(TRUE, "bmp", ".bmp", OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT, "
        位图文件(*.bmp)|*.bmp|JPEG 文件(*.jpg)|*.jpg|GIF 文件(*.gif)|*.gif||");
    if (dlg1.DoModal() == IDOK)
    {
        if (hwnd[1] != NULL)
            hwnd[1] = NULL;
        if (hSrcDC[1] != NULL)
            hSrcDC[1] = NULL;
        if (hDesDC[1] != NULL)
            hDesDC[1] = NULL;
        hwnd[1] = GetDlgItem(IDC_STATIC1);
        hDesDC[1] = hwnd[1] -> GetDC() -> m_hDC;
        hSrcDC[1] = CreateCompatibleDC(hDesDC[1]);
        watermark = dlg1.GetPathName();
        hBitmap[1] = (HBITMAP) LoadImage(AfxGetInstanceHandle(), watermark, IMAGE_BITMAP,
0, 0, LR_LOADFROMFILE | LR_CREATEDIBSECTION);
        GetObject(hBitmap[1], sizeof BITMAP, &bm[1]);

        SelectObject(hSrcDC[1], hBitmap[1]);
        hwnd[1] -> GetClientRect(&rect[1]);
        ::SetStretchBltMode(hDesDC[1], COLORONCOLOR);
        ::StretchBlt(hDesDC[1], rect[1].left, rect[1].top, rect[1].right, rect[1].
bottom, hSrcDC[1], 0, 0, bm[1].bmWidth, bm[1].bmHeight, +SRCCOPY);
        show[1] = TRUE;

        if ((bm[0].bmHeight * bm[0].bmWidth) < 2 * (bm[1].bmHeight * bm[1].bmWidth))
        {
            //不能嵌入水印
            MessageBox("水印图像过大请更换较大宿主图像", "错误", MB_OK);
            Mark = FALSE;
        }
        else
        {
            SetTimer(NULL, 50, 0);
            //WaterBitmap2Data()将图像转换为数据保存在二维数组 m_watermarkdata 中
            POSITION pos = dlg1.GetStartPosition();
            watermark = dlg1.GetNextPathName(pos);
            WaterBitmap2Data(); //GetDlgItem(IDC_STOP_BUTTON) -> EnableWindow(FALSE);
            Mark = TRUE;
            GetDlgItem(IDC_BUTTON3) -> EnableWindow(TRUE);
        }
        CString s, s0, s1, s2;
        s.Format("%2d", bm[1].bmHeight); //将水印图像的高转化为二进制
        s1.Format("%2d", bm[1].bmWidth); //将水印图像的宽转化为二进制
        s0.Format("%d", m_dib1 -> GetBiBitCount()); //将水印图像的计数转化为十进制
    }
}

```

```

s2="水印图像"+s0+"位 H×W:"+s+"X"+s1;

if(pEdit1!=NULL)
    pEdit1=NULL;
r1.left=rect[0].right+35;r1.top=rect[1].bottom+2;r1.right=200+r1.left;r1.
bottom=r1.top+20;
pStatic1 = new CStatic;
pStatic1->Create(s2,ES_CENTER|WS_VISIBLE|ES_READONLY,r1,this,4);
}
}

```

WaterBitmap2Data()与 Bitmap2Data()将宿主图像转换为二维数组数据的方法大体是相同的,其中主要的区别是 WaterBitmap2Data()函数会判断载入图像是否为小于等于 8 位的灰度图,如果不是的话,将无法进行水印的嵌入。WaterBitmap2Data()函数体如下:

```

void CWaterMarkDlg::WaterBitmap2Data()//将水印图像转换为数据保存在二维数组 m_watermarkdata 中
{
    int i,j;
    if(m_watermarkdata !=NULL)
        if(m_Wh!=0)
            for(i=0; i<m_Wh; i++)
                delete [] m_watermarkdata[i];
        else
            return ;
    if(watermark.GetLength()==0)
    {
        MessageBox("无效的水印文件");
        return ;
    }
    m_dib1->Open(watermark);if(m_dib1->GetBiBitCount()>8) MessageBox("目前仅对小于等于 8 位的水印信息有效");
    WaterWidth=m_dib1->GetWidth();           //获取水印图像的宽
    WaterHeight=m_dib1->GetHeight();          //获取水印图像的高
    biBitCount1 = m_dib1->GetBiBitCount();     //获取水印图像的计数值
    BYTE *colorTable;
    colorTable =(BYTE *) m_dib1->m_pDibBits;   //获得水印图像的彩色表
    int byteBitCount = m_dib1->GetBiBitCount()/8;
    m_watermarkdata =new RGBQUAD*[WaterHeight]; //定义二维数组的大小
    for (int l=0;l<WaterHeight;l++)
        m_watermarkdata[l]=new RGBQUAD [WaterWidth];
    int count = 0;
    int num = 0;
    //分别对二维数组中的每个元素的 RGB 值进行赋值操作
    for(i=WaterHeight-1; i>=0; i--)
    {
        for(j=0; j<WaterWidth; j++)

```



```

{
    m_watermarkdata[i][j].rgbBlue = colorTable[count++];
    m_watermarkdata[i][j].rgbGreen = colorTable[count++];
    m_watermarkdata[i][j].rgbRed = colorTable[count++];
    m_watermarkdata[i][j].rgbReserved = 0;
    count += byteBitCount-3;
    num+=1;
    //TRACE("\n@%d @%d @%d",
    //m_watermarkdata[i][j].rgbRed,
    //m_watermarkdata[i][j].rgbBlue,
    //m_watermarkdata[i][j].rgbRed);
}
count += (4-(WaterWidth*byteBitCount)%4)%4;
}
}

```

嵌入水印的函数名为 OnWaterEmbed(), 这个函数主要是实现能否嵌入水印图像到宿主图像的再次判断、定义一个新的图像数据副本并加载宿主图像、将水印图像嵌入到宿主图像副本之中以及将嵌入好的数据图像打印在程序之中。这里的嵌入思想是获取宿主图像当前像素点 RGB 的 Red 值并保存在 m\_r 中, 获取宿主图像当前像素点 RGB 的 Blue 值并保存在 m\_b 中, 获取水印图像当前像素点 RGB 的 Green 值并保存在 c 中, 然后通过 Operate\_Byte() 方法, 将 m\_r 和 m\_b 的值用 c 进行一定的改变, 将改变完成后的每个像素点 RGB 中 Red 和 Blue 值赋给水印嵌入后的位图中对应每个像素点 RGB 中 Red 和 Blue 的值。OnwaterEmbed() 函数体如下:

```

void CWaterMarkDlg::OnWaterEmbed()
{
    // TODO: Add your control notification handler code here
    if ((bm[0].bmHeight*bm[0].bmWidth)<3*(bm[1].bmHeight*bm[1].bmWidth))
    {
        //不能嵌入水印
        MessageBox("水印图像过大请更换较大宿主图像", "错误", MB_OK);
        Mark=FALSE;
    }
    if(!Mark)
        return ;
    if(biBitCount<biBitCount1)
    {
        MessageBox("水印受限制");
        return ;
    }
    RGBQUAD *m_copymater; //定义一个水印信息的数据副本
    m_copymater = new RGBQUAD [WaterHeight*WaterWidth];
    int i,j;
    int count=0;
    BYTE m_r,m_b;

```

```

if(m_waterEmbed !=NULL)          //m_waterEmbed 是水印嵌入后的位图数据全局变量
    if(m_Ih!=0)
        for(int l =0;l<m_Ih;l++)
            delete [] m_waterEmbed[l];
m_waterEmbed = new RGBQUAD*[ImageHeight];
for(i=0;i<ImageHeight;i++)
    m_waterEmbed[i] = new RGBQUAD [ImageWidth];
for(i=0;i<ImageHeight;i++)        //将原来的宿主图像的数据复制到 m_waterEmbed 里
    for(j=0;j<ImageWidth;j++)
        m_waterEmbed[i][j]=m_tOriPixelArray[i][j];
        //将水印信息的数据复制到一维数组 m_copymater 水印信息的数据副本里
for(i=0;i<WaterHeight;i++)
    for(j=0;j<WaterWidth;j++)
    {
        m_copymater[i*WaterHeight+j]=m_watermarkdata[i][j];
        count++;
    }
number=count;//记录嵌入的点数
int k=0;
if (biBitCount==24)                //Operate_Byte() 嵌入水印信息位操作
{
    for(i=0;i<ImageHeight&&count>0;i++)
        for(j=0;j<ImageWidth&&count>0;j++,count--)
        {
            //获取宿主图像当前像素点 RGB 的 Red 值并保存在 m_r 中
            m_r=m_waterEmbed[i][j].rgbRed;
            //获取宿主图像当前像素点 RGB 的 Blue 值并保存在 m_b 中
            m_b=m_waterEmbed[i][j].rgbBlue;
            BYTE &a=m_r;
            BYTE &b=m_b;
            //获取水印图像当前像素点 RGB 的 Green 值并保存在 c 中
            BYTE c=m_copymater[i*ImageHeight+j].rgbGreen;
            //调用 Operate_Byte 函数实现位操作以完成水印嵌入
            Operate_Byte(a,b,c);
            //将操作后的当前像素点 RGB 中 Red 值赋给水印嵌入后的
            //位图中当前像素点 RGB 中 Red 的值
            m_waterEmbed[i][j].rgbRed=m_r;
            //将操作后的当前像素点 RGB 中 Blue 值赋给水印嵌入后的
            //位图中当前像素点 RGB 中 Blue 的值
            m_waterEmbed[i][j].rgbBlue=m_b;
            //TRACE("\n%d",c);k++;
        }
    PutInWaterMessage();//
}
else
    return;

//TRACE("\n 次数是%d",k);

```

```

CDC *dc=GetDC();           //将像素打印出来
RECT m_rect;
float w,h,p,q;
GetDlgItem(IDC_STATIC2)->GetClientRect(&m_rect);
if(m_rect.right<ImageWidth&& m_rect.bottom<ImageHeight)
{
    w=(float)m_rect.right/(float)ImageWidth;
    h=(float)m_rect.bottom/(float)ImageHeight;
}
else
    w=h=1;
for( p=0,i=0;i<ImageHeight;i++,p++)
    for( j=0, q=0;j<ImageWidth;j++,q++)

dc->SetPixel(2+(int)(q*w),235+(int)(p*h),RGB(m_waterEmbed[i][j].rgbRed,
        m_waterEmbed[i][j].rgbGreen,m_waterEmbed[i][j].rgbBlue));
ReleaseDC(dc);
delete [] m_copymater;
m_Ih=ImageHeight;          //为保证直接更换水印信息而没有更换宿主图像提供原宽和高
m_Iw=ImageWidth;
m_Wh=WaterHeight;
m_Ww=WaterWidth;
TRACE("\n%d\n",ImageHeight);
GetDlgItem(IDC_BUTTON4)->EnableWindow(TRUE);
GetDlgItem(IDC_BUTTON5)->EnableWindow(TRUE);
}

```

接下来重点描述一下 Operate\_Byte() 这个函数,这也是嵌入水印的核心代码,在 Operate\_Byte() 函数中,三个形参中前 2 个参数的值分别是宿主图像中每个像素点的 Red 和 Blue 值,第三个是水印图像中相应像素点 RGB 中 Green 的分量值。首先将 m\_operate 中的值以二进制的形式存放到一个长度为 8 的一维数组 x 之中,使得 x 数组每一位存放的就是水印图像中当前像素点下长度为 8 的 Green 值的二进制的每一位数值 0 或者 1。例如 m\_operate 的值为 241,则当前 x 数组中值为 11110001。之后分别将 x 数组中前 4 位和后 4 位分别存放在 H 和 L 中,此时 H 和 L 中的数值最高为 7,最低为 0,之所以这样是因为 H 和 L 分别将 x 中的值进行了一半的截取,而又采用的是二进制,所以存放的数值只是一个最大为 7 最小为 0 的数值。这时,分别将 operate1 和 operate2 (存放的是宿主图像每个当前像素点 RGB 中 Red 和 Blue 的分量值)与 M 进行与操作,使得 operate1 和 operate2 的高四位保持原状,而低四位变为 0000。之后再获得的水印图像每个当前像素点 RGB 中 Green 值的高四位值存放到 operate1 中的低四位之中,将 Green 值的低四位存放到 operate2 中的低四位中,从而完成水印的嵌入。Operate\_Byte() 函数体如下:

```

void CWaterMarkDlg::Operate_Byte(BYTE &operate1, BYTE &operate2, BYTE m_operate)
{
    //将 m_operate 最低四位取出来放到 L 里,最高四位放到 H 里,

```

```

//最后将 H 赋值给 operate1 最低四位, L 赋值给 operate2 最低四位
int i;
BYTE M=240,t=m_operate;//1111 0000
BYTE H,L,x[8],y[4];
y[3]=8;y[2]=4;y[1]=2;y[0]=1;
//将 m_operate 的二进制表示存入到 x 数组中
for(i=0;i<8;i++)
{
    x[i]=m_operate&1;
    m_operate>>=1;
}
H=x[7]*y[3]+x[6]*y[2]+x[5]*y[1]+x[4]*y[0]; //将 m_operate 中高四位的值与 1 存入 H
L=x[3]*y[3]+x[2]*y[2]+x[1]*y[1]+x[0]*y[0]; //将 m_operate 中低四位的值与 1 存入 L
operate1&=M; //将 operate1 总的值与 M 进行与操作从而将 operate1 变为 xxxx 0000
operate2&=M; //将 operate2 总的值与 M 进行与操作从而将 operate2 变为 xxxx 0000
operate1+=H; //将 operate1 的值与高四位 H 的值进行相加
operate2+=L; //将 operate2 的值与低四位 L 的值进行相加
}

```

嵌入水印操作完成后伴随着是水印的提取,提取是水印嵌入的反操作,分别将水印后的图像当前像素点的 RGB 值分量高四位和低四位分离出来,分离出之后将二者拼接为一个新的长度为 8 的二进制数,这个新的二进制数(程序中用字节数组表示)转化为十进制数之后就是当前像素点下水印图像 RGB 值的 Green 分量。所以不做过多说明,Operate\_ByteOut()函数体如下:


```

BYTE CWaterMarkDlg::Operate_ByteOut(BYTE operate1, BYTE operate2)
{
    int i = 0;
    BYTE x[8],y[8],data;
    for (i=0;i<4;i++)
    {
        x[i]=operate2&1;
        operate2>>=1; //取出最低的四位
    }
    for (i=4;i<8;i++)
    {
        x[i]=operate1&1;
        operate1>>=1; //取出最高的四位
    }
    y[0]=1;y[1]=2;y[2]=4;y[3]=8;y[4]=16;y[5]=32;y[6]=64;y[7]=128;
    data=x[0]*y[0]+x[1]*y[1]+x[2]*y[2]+x[3]*y[3]+x[4]*y[4]+x[5]*y[5]+x[6]*y[6]
        +x[7]*y[7];
    return data;
}

```

值得注意的是,在嵌入与提取水印时,只选取了水印每个像素点的 RGB 中 Green 值作为变量,这是因为 BMP 图像在转换为灰度图像时绿色所占的比重最大,所以在转换时直接使用 G 值,

这也是进行水印嵌入与提取的一个重要依据。

 BMP 图像在转换为灰度图像时的公式为  $\text{Gray}(i,j)=0.11 \times R(i,j)+0.59 \times G(i,j)+0.3 \times B(i,j)$

### 6.4.2 最低有效位算法实现

数字图像水印系统中基于最低有效位水印的算法是基本 LSB 算法的一种变形,但主要思想还是替换最低有效位,只不过是在处理之前进行了一次异或操作,增加了系统的鲁棒性。对于宿主图像以及水印图像的载入和保存操作,分别存放在了 DIB 类中 Load()和 Save()函数中,大致与不带嵌入因子的加性规则算法相似,这里不再详细介绍,只对嵌入和提取函数进行详细阐述。Bmdata 是宿主图像当前像素点的数据,efdate 是水印图像当前像素点的数据。分别将宿主和水印图像当前像素点的数据保存到数组 x 和 s 之中, x 和 s 都是长度为 8 的一维数组。而当前 x 数组的第一个位置存放的就是宿主图像当前像素点的最低有效位,接下来将水印图像的第 7 位与宿主图像的最低位进行异或操作,获取出新的隐藏信息,之后将该信息重新赋值给宿主图像的最低位,从而将其值改变。在进行提取时,只需进行相应的反操作即可。Embed()函数体如下:

```
void CDib::Embed()                                //嵌入
{
    unsigned char bmdata;                          //宿主图像数据
    unsigned char efdata;                          //水印后图像数据
    int t = 7;
    int x[8];
    int s[8];
    int last_bit;                                  //记录字节最低位本来的 bit
    for(UINT i1 = 0, i2 = 0; i1 <= bitmap_size - 1, i2 <= embfile_size - 1; i1++)
    {
        bmdata = *p;
        for (int j = 0; j <= 7; j++)                //计算各 bit 位
        {
            x[j] = bmdata & 1;
            bmdata >>= 1;
        }
        last_bit = x[0];
        x[0] = x[1] ^ x[2] ^ x[3] ^ x[4] ^ x[5] ^ x[6] ^ x[7];
        if (t == 7)                                  //宿主图片每走过 8 个字节, 计算一次 s[]
        {
            efdata = *q;
            for (int j = 0; j <= 7; j++)
            {
                s[j] = efdata & 1;
                efdata >>= 1;
            }
        }
    }
}
```





```

t--;
if (t == -1)                                //s[7]到s[0]组成一个字节
{
    *q = s[7] * 128 + s[6] * 64 + s[5] * 32 + s[4] * 16 +
        s[3] * 8 + s[2] * 4 + s[1] * 2 + s[0];
    t = 7;
    i2++;
    q++;
}
p++;
}
}

```

## 6.5 经验分享

本章主要讲述了数字图像水印系统的核心原理、功能、流程、嵌入和提取的一般方法。在编程实现阶段中用两个门槛较低的嵌入和提取方法向读者展示了如何实现不可见水印的嵌入与提取。不难看出，数字图像水印技术的主要流程包括了水印生成、嵌入、检测和提取这4个环节，每一个环节又都与其他环节有连带关系。从具体细节来看，本章中的数字图像水印系统两种方法主要考虑角度都从载体与水印图像的像素点上来处理。不带嵌入因子的简单加性法则算法利用的原理是人眼对红色和蓝色不如对绿色的敏感度高，且由彩色图像转成的灰度图像时，原绿色分量所占比例是RGB三原色之中最大的。因此提取水印图像绿色分量(G)的高四位和低四位分别嵌入到宿主图像红色(R)和蓝色(B)分量的低四位中，对嵌入水印后图像的视觉效果基本不会有影响，但同时又保持了原水印图像的主要信息，思想非常巧妙。而最低有效位的算法在数字图像水印系统中也算是最基本、最可靠的方法了，虽然其鲁棒性、安全性特别差，但是其最低有效位的思想确实很值得研究。此外本程序仅限于同一平面上的最低有效位，其实可以拓展到多个平面的，这样其鲁棒性和安全性能更高，抗攻击能力可以更强。从目前的研究状况来看，60%的数字水印技术都是针对图像载体展开的，但是无论针对载体还是针对嵌入体，无论是从水印的鲁棒性还是安全性，都有许多悬而未决的问题，因此，对该领域的研究仍将持续较长一段时间。

数字图像水印系统是包含两个算法的系统，一种是不带嵌入因子的加性规则算法，另外一种是最有效位的算法。为了便于读者阅读，将两种算法代码分别写在两个流程之中，可以统一或者独立运行。读者在进行编译运行时，先在debug文件夹中找到WaterMark\_LSB工程，编译并运行最低有效位算法，之后再在WaterMark文件夹中编译WaterMark工程。此外，数字图像水印系统的工作环境是VC 2008，处理器是32位，所以读者如果是64位机器请注意修改变量长度。运行时如果出现缺少dll文件等错误请读者在项目属性中找到连接器里的调试，将其中的生成映射文件选为“是”即可。

## 第 7 章 遥感图像配准系统

在各地的妈祖庙里，总能看到两尊极为独特的神祇，他们衣着裸露，一位一手拿叉一手遮于眉上，像在远眺，另一位一手拿戟一手遮于耳后，像在倾听，他们都是妈祖的侍神，远眺的那位就是传说中的千里眼。科技的发展让这个古老的神话变成了现实，遥感便是信息时代的千里眼。

遥感一般是指从远离地面的空间工作平台上，通过传感器对地球表面的电磁波信息进行探测，并经信息的传输、处理与分析，对地球的资源与环境进行探测和监测的综合性技术，已广泛应用于农业、林业、地质、海洋、气象、水文、军事、环保等领域。遥感图像是由航空、航天或接近地面等手段所获取的光谱资料，根据处理手段的不同，可分为光学处理和数字处理。遥感图像数字处理就是将遥感图像的模拟或数字形式的信息输入计算机中，利用某种遥感图像处理软件，按照一定的数学模型进行变换、加工，产生可为专业人员判读的图像或数据。遥感数字图像处理一般包括图像的转换、校正、增强、多源信息融合（配准）、解译等操作。本章主要以开发一个简单的遥感图像配准系统为例，探讨遥感图像的校正、增强和配准的原理及编程实现方法。

### 本章要点：

- 遥感图像几何校正技术
- 遥感图像辐射校正技术
- 遥感图像增强技术
- 遥感图像配准技术
- 遥感图像配准系统功能描述
- 遥感图像配准系统的总体结构和主要流程
- 遥感图像配准系统的编程实现

### 7.1 核心技术原理

遥感图像配准系统涉及的核心技术主要包括图像几何校正与辐射校正技术、图像增强技术和图像配准技术。

### 7.1.1 遥感图像几何校正技术

几何校正是遥感图像处理的一个重要环节，是削弱遥感图像与地面真实形态差异的重要手段。

几何校正和几何畸变是遥感理论的一对派生词，几何校正是因几何畸变而产生，是解决几何畸变的方法体系。在遥感理论上，将遥感平台位置和运动状态、地形起伏、地球表面曲率和大气折射等遥感系统内外因素影响造成的遥感图像几何位置上的变化统称为几何畸变，也就是遥感图像在几何位置上与实际地面位置有差异。在图像上表现为位移、旋转、缩放、仿射、弯曲和更高阶的歪曲，像元行列分布不均匀，像元大小与地面大小对应不准确。

遥感图像几何校正按照处理方式分为光学校正和数字校正两大类。光学校正通常不能对卫星遥感图像，特别是动态遥感图像进行严格的校正，而数字校正是建立在严格的数学基础上的，并可以逐点地对影像进行校正，所以原则上它可以对任何类型的传感器影像进行严格的校正。其中，数字校正主要是通过影像元素坐标的解析变换和影响灰度值的重抽样过程来获取离散结构的新数字影像。数字校正有以下两种处理方法。


#### 1. 系统性校正

系统性校正用来处理系统畸变。由于系统畸变可以用严格的数学表达式来描述，因此，当消除图像几何畸变的理论校正公式已知时，可把该式中所含的与遥感器构造有关的校准数据（焦距等）及遥感器的位置、姿态等的测量值代入到理论校正公式中进行几何校正。

#### 2. 非系统性校正

非系统性校正用来处理随机畸变。与系统畸变相反，随机畸变难以用明确的数学表达式来描述。消除此类畸变的主要方法有多项式法和共线方程法等。其中，多项式法的机理是通过若干GCP（Ground Control Point），建立不同图像间的多项式空间变换和像元插值运算，实现遥感图像与实际地理图件间的配准，达到消减以及消除遥感图像几何畸变的目的。

本系统将通过多项式法实现遥感图像的几何校正。

 多项式法用途很广，在遥感图像制图、几何配准和复合分析中都有重要应用。在 7.1.4 节中还会用到多项式法实现遥感图像的几何配准。

多项式几何校正机理实现包括两步。

#### 步骤一：图像坐标的空间变换。

判断一幅图像有没有几何畸变，一个重要的特征就是观察该图像对应像元的坐标。图 7-1b 为无几何畸变的图像像元分布图，像元是均匀且等距分布的；图 7-1a 为有几何畸变的遥感图像像元分布图，像元是非均匀且不等距分布的。因此，为了在有几何畸变的图像上获取无几何畸变的像元坐标，需要对这两个图像的坐标系统进行空间转换。

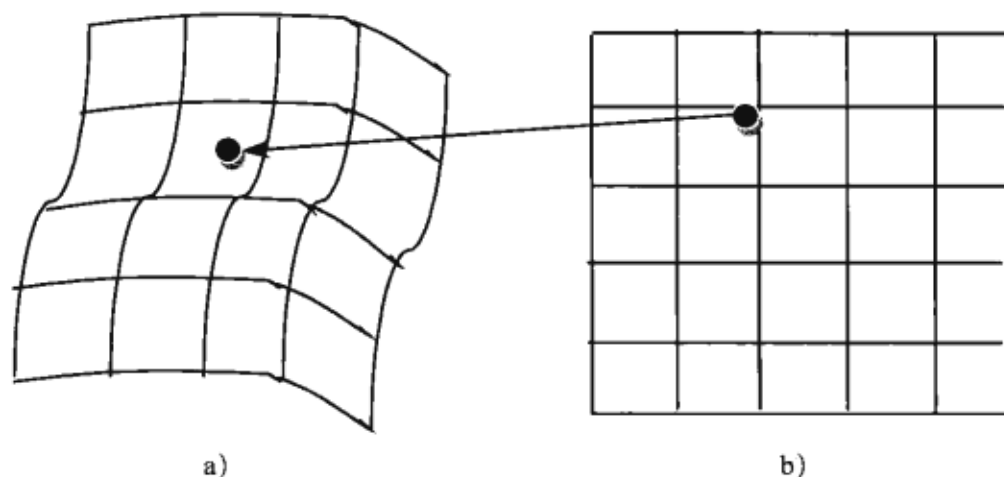


图 7-1 遥感图像几何畸变示意图

在数学方法上，对于不同二维坐标系间的空间转换，通常采用的是二元  $n$  次多项式。

$$\begin{cases} x = \sum_{i=0}^n \sum_{j=0}^{n-i} a_{ij} u_i v_j \\ y = \sum_{i=0}^n \sum_{j=0}^{n-i} b_{ij} u_i v_j \end{cases} \quad (7-1)$$

二元  $n$  次多项式将不同坐标系下的对应点坐标联系起来， $(x, y)$  和  $(u, v)$  分别对应不同坐标系中的像元坐标。这是一种多项式数字模拟坐标变换的方法，一旦有了该多项式，就可以从一个坐标系推算出另一个坐标系中的对应点坐标。

如何获取和建立二元  $n$  次多项式，即二元  $n$  次多项式系数中  $a$  和  $b$  的求解，是几何校正成败的关键。数学上有一套完善的计算方法，核心是通过已知若干存在于不同图像上的同名点坐标，建立求解  $n$  次多项式系数的方程组，采用最小二乘法得出二元  $n$  次多项式系数。

在二元  $n$  次多项式数字模拟中，从提高几何校正精度的角度考虑，需要兼顾的因素主要有引起几何畸变的原因和产生数学运算误差的因素。归纳起来有以下 3 个方面的考虑因素：

- 多项式中  $n$  值的选择。当  $n=1$  时，上述的坐标空间变换成二元一次多项式，可以进行线性的坐标变换，解决比例尺、中心移动、歪斜等方面的几何畸变。当  $n \geq 2$  时，上述的坐标空间变换成为二元非线性多项式，解决遥感器偏航、俯仰、滚动等因素引起的几何畸变。理论上讲， $n$  值越大，越能校正复杂的几何畸变，但计算量也相对要大。实际应用中通常取  $n \leq 3$ 。
- GCP 的选择。比较成熟的 GCP 选择方法是：通过目视，选择熟悉的、易分辨且精度较高的特征点（如小水塘边缘、线状地物的交叉点、海岸线弯曲处等），图像边缘部分一定要选取 GCP，以避免外推。同时使采集的 GCP 均匀分布于整个图像上。
- 最后是 GCP 数目的确定。从数学运算上来说，一次多项式变换，需要 GCP 的最少数目是

3. 二次多项式变换, 需要 GCP 的最少数目是 6。  $n$  次多项式, 需要 GCP 的最少数目为  $(n+1)(n+2)/2$ 。

### 步骤二: 图像像元灰度值重采样。

经过上述图像像元坐标的空间变换, 得到对应于实际地面或无几何畸变的图像坐标, 图像上每个像元都有无几何畸变的坐标值。随后需要做的是给每个像元赋灰度值, 因为已知的图像数据是有几何畸变的像元灰度值, 并没有校正后的无几何畸变的像元灰度值。

校正前后图像的分辨率变化和像元点位置相对变化引起输出图像阵列中的同名点灰度值变化。如图 7-2 所示, 图 7-2a 为校正后图像像元  $P$ , 图 7-2b 为未校正图像  $P$  点的同名点  $P'$ ,  $P$  点的灰度值与  $P'$  应该一致, 然而未校正图像中  $P'$  点的灰度值无从得知,  $P$  点的灰度值也就无法确定。

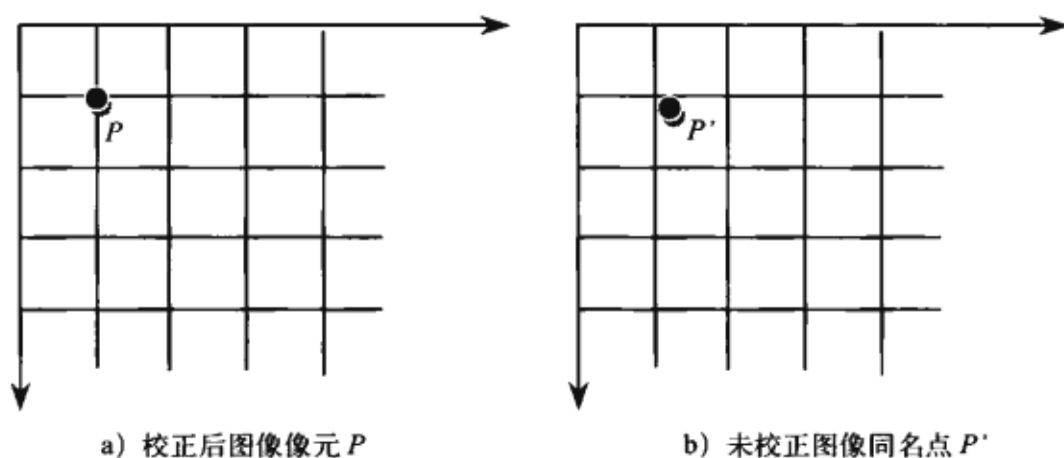


图 7-2 遥感图像校正前后变化示意图

所以需要对遥感图像灰度值进行重采样。  $P'$  的灰度值取决于周围阵列点上像元的灰度值对其所做的贡献, 计算出校正后像元位置的灰度值, 就形成了无几何畸变的遥感数据。重采样的方法有最近邻法、双向线性内插法和三次卷积内插法等。

1) 最近邻法。用距离投影点最近的像元灰度值代替输出像元灰度值。这种方法的优点是: 保留大量原始灰度值, 没有经过平滑处理, 易于区分线性地物, 简易省时, 适用于专题文件。它的缺点是: 锯齿状, 不平滑, 可能出现不连续。

2) 双向线性内插法。这种方法的优点是: 平滑, 没有锯齿状; 与最近邻法相比, 空间信息更准确; 常用于改变像元大小, 如数据融合。缺点是: 像元值被平均化, 可能导致某些边缘信息丢失。

3) 三次卷积内插法。获取与投影点邻近的 16 个像元灰度值计算输出像元灰度值。这种方法的优点是: 与其他方法相比, 均值和标准偏差与原始像元一致; 可以锐化图像, 平滑噪声。缺点: 数据值可能改变, 计算费时。

### 7.1.2 遥感图像辐射校正技术


遥感成像时,由于各种因素的影响,使得传感器观测目标的反射或辐射能量时,所得到的测量值与目标的光谱反射率或光谱辐射亮度等物理量之间存在差异,其间的差值称为辐射误差。辐射校正就是为消除遥感图像的辐射误差而进行的校正。

遥感图像产生辐射误差的因素主要有:

- 1) 大气传输(雾和云等)条件。
- 2) 太阳照射(位置与角度)。
- 3) 地形起伏引起的辐射强度变化。
- 4) 传感器响应特性。
- 5) 影像处理,如摄影处理等。影像灰度失真与影像空间频率有关,空间频率越高,即目标越小时,辐射误差越大。

而针对不同辐射误差产生的原因,辐射校正可分为以下几类:

- 1) 大气辐射误差校正。有3种方法:统计学方法、辐射传递方程计算法、波段对比法。
- 2) 太阳辐射校正。此种校正就是校正由太阳高度角引起的辐射畸变,即将太阳光线倾斜照射时获取的图像校正为垂直照射时获取的图像。其校正方法有:利用太阳高度和太阳高度角进行校正,利用波段比值校正。
- 3) 地形起伏引起的辐射误差校正。校正方法有:利用地表法线矢量与太阳入射矢量两者的夹角校正,利用多波段比值校正。
- 4) 传感器引起的辐射误差校正。此校正通常采用内部校准光源和校准楔,如陆地卫星多光谱扫描仪的辐射校正。
- 5) 影像辐射畸变校正。常采用物理或数学(校正曲线或各种算法)方法,如空间滤波、平滑化,校正各种灰度失真及斑点、灰点、条纹、信号缺失等分布在整个影像上的离散形式的辐射误差。

 一般遥感图像数字处理在进行辐射校正时,主要完成大气辐射误差校正和地形起伏引起的辐射误差校正两部分。由于相关参数数据的缺失,且图像辐射误差相对较小,本章不再对遥感图像进行辐射校正。

### 7.1.3 遥感图像增强技术

遥感图像增强作为基本的图像处理技术,其主要目的有两个:一是改善图像的视觉效果,提高图像的清晰度;二是针对给定图像的应用场合,强调某些感兴趣的特征,抑制不感兴趣的特征,以扩大图像中不同物体特征之间的差别,满足某些特殊分析的需要。



本系统使用以下 5 类遥感图像增强技术。

### 1. 直方图增强

图像直方图是图像处理中一种十分重要的图像分析工具，它描述了一幅图像的灰度级内容，任何一幅图像的直方图都包含了丰富的信息。图像直方图灰度级的分布形态可以提供图像信息的许多特征，所以可以通过改变直方图的形状来达到增强图像对比度的效果。常用的方法有直方图均衡化与直方图规定化。

### 2. 灰度变换增强

图像的灰度变换增强是图像增强处理技术中一种简单、直接的基于空间域的图像处理方法。灰度变换主要针对独立的像素点进行处理，由输入像素点的灰度值来决定相应的输出像素点的灰度值，通过改变原始图像数据所占据的灰度范围而使图像在视觉上得到改观。灰度变换增强方法又分为线性灰度增强、分段线性灰度增强与非线性灰度增强。

### 3. 图像平滑处理

图像平滑是一种实用的图像处理技术，能消除图像采集、传输、处理过程中的噪声。图像平滑包括空域法和频域法两大类，在空间域中主要利用邻域平均法、加权平均法、选择式掩膜平滑法和中值滤波法来消除图像噪声；在频率域中主要利用各种形式的低通滤波器来消除噪声。

### 4. 图像锐化处理

图像锐化就是补偿图像的轮廓，增强图像的边缘及灰度跳变的部分，使图像变得清晰。图像平滑往往使图像中的边界、轮廓模糊，为了减少这类不利效果的影响，就需要利用图像锐化技术使图像的边缘清晰。图像锐化处理的目的是为了使图像的边缘、轮廓线以及图像的细节变得清晰，经过平滑的图像变得模糊的根本原因是对图像进行了平均或积分运算，因此可以对其进行逆运算（如微分运算）就可以使图像变得清晰。图像锐化的两种常用方法分别是：梯度锐化和拉普拉斯掩膜锐化。

### 5. 伪彩色增强

伪彩色增强是把灰度图像的各个不同灰度级按照线性或者非线性的映射函数变换成不同的彩色，使图像细节更容易辨认，目标更容易识别。

由于篇幅原因，此节只对遥感图像增强技术进行概述，请读者参考由刘海波、沈晶、郭耸等编著的《Visual C++数字图像处理技术详解》一书（机械工业出版社出版）。该书第 5 章给出了图像增强相关的步骤和具体的编程实现。

## 7.1.4 遥感图像配准技术


几何配准是将不同时间、不同波段、不同遥感器系统所获得的同一地区的图像，经几何变换

使同名像点在位置上和方位上完全叠合的操作。

随着遥感技术的飞速发展,人们对遥感数据的需求也呈现多元化,它们可能来自不同的时间、不同的波段、不同的传感器系统。在应用这些多时相与多信息的复合图像数据来进行计算机自动分类和地物特征的变化监测或其他复合综合分析应用处理时,必须保证各不同图像间的几何一致性,即需要进行图像间的几何配准。

校正和影像配准原理基本是一样的,几何校正是借助一组地面 GCP,对一幅图像进行地理坐标的校正,把影像纳入一个投影坐标系中,有坐标信息地理参考;影像配准是用一影像对另一幅图像的校准,在不同影像上选取同名点作为 GCP,一式两幅图像进行同名像元配准。

---

 为了提高配准精度,本系统将添加半自动方式来选取 GCP,即在基准图像中选取一个 GCP,以此 GCP 为中心提取像素模板,在待配准图像中全局搜索同名点。

---

## 7.2 系统功能

本系统的主要功能就是对遥感图像进行几何配准,为了达到更好的配准效果,在配准前对遥感图像进行几何校正和图像增强处理,下面具体描述一下该系统的功能以及界面效果。

### 7.2.1 功能描述

该系统主要实现以下功能:

- 1) 可以对主窗口图像进行几何校正(手动选择 GCP),将结果保存并在主窗口中打开,方便下一步操作。
- 2) 可以对主窗口图像进行图像增强,包括:直方图增强(直方图均衡化、直方图规定化)、灰度变换增强(线性灰度增强、分段线性灰度增强、非线性灰度增强)、平滑处理(邻域平均法、加权平均法、选择式掩膜平滑法、中值滤波法)、锐化处理(梯度锐化、拉普拉斯掩膜锐化)、伪彩色增强,将结果保存并在主窗口中打开,方便下一步操作。
- 3) 可以对主窗口图像进行图像配准,选取 GCP 时有两种选择方式:手动和半自动,为了得到更高的配准精度,优先使用半自动的方法选取 GCP,如果半自动方法选取不准确,则改用手动方式选取 GCP。图像配准后将结果保存并在主窗口中打开。

### 7.2.2 界面效果

遥感图像几何校正效果如图 7-3 所示,图像增强效果如图 7-4 所示,图像配准效果如图 7-5 所示。

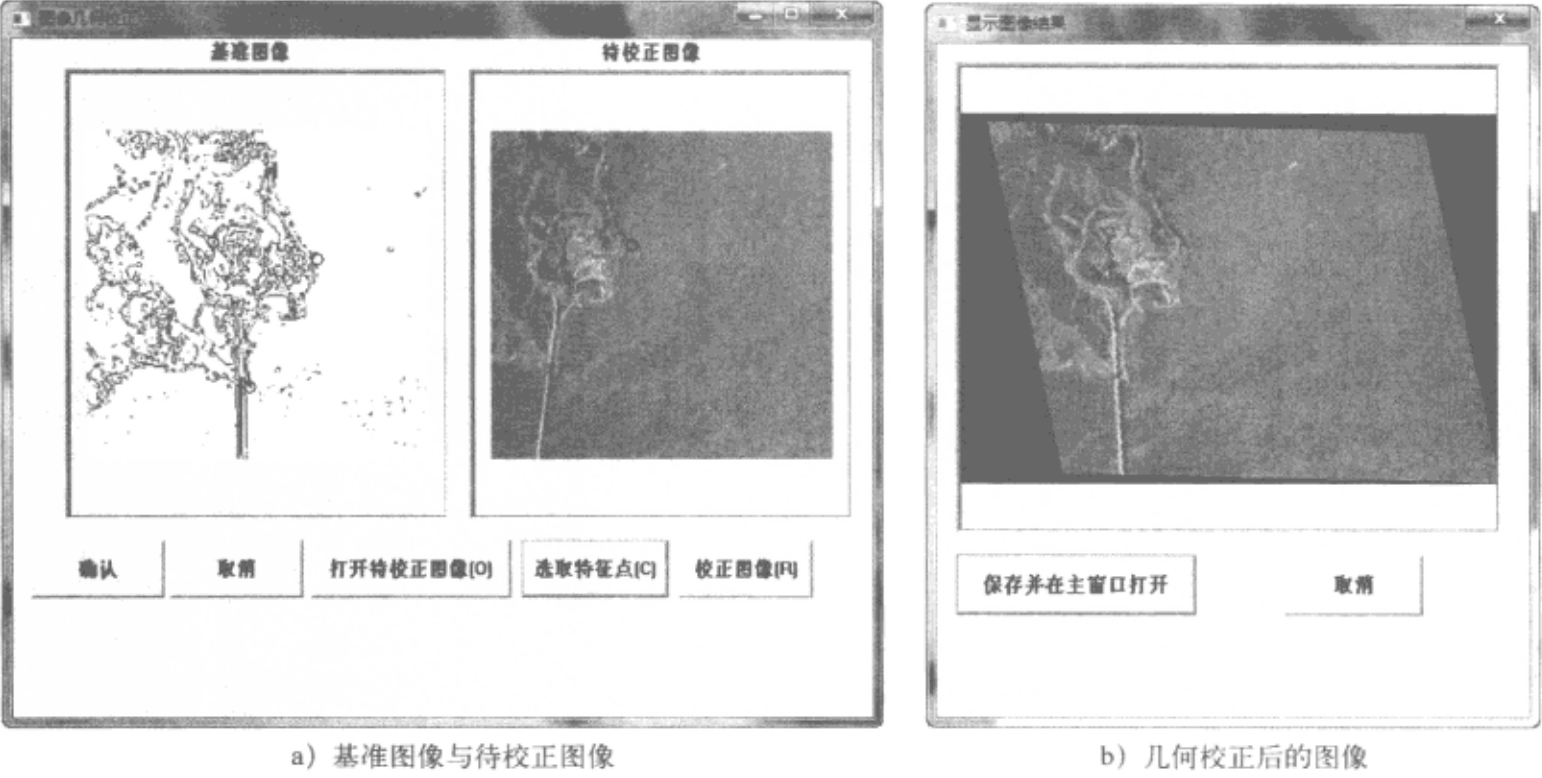


图 7-3 几何校正效果

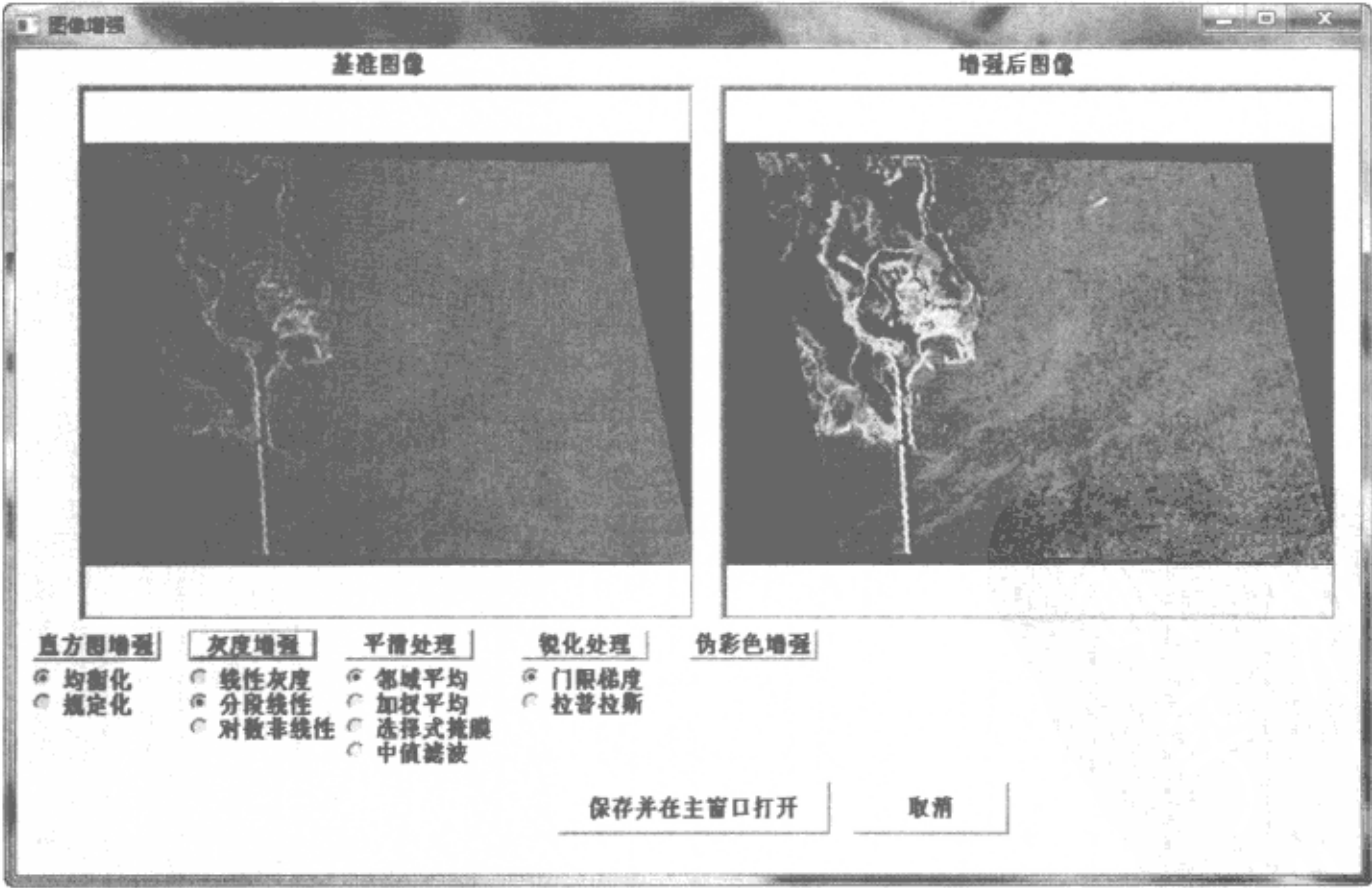
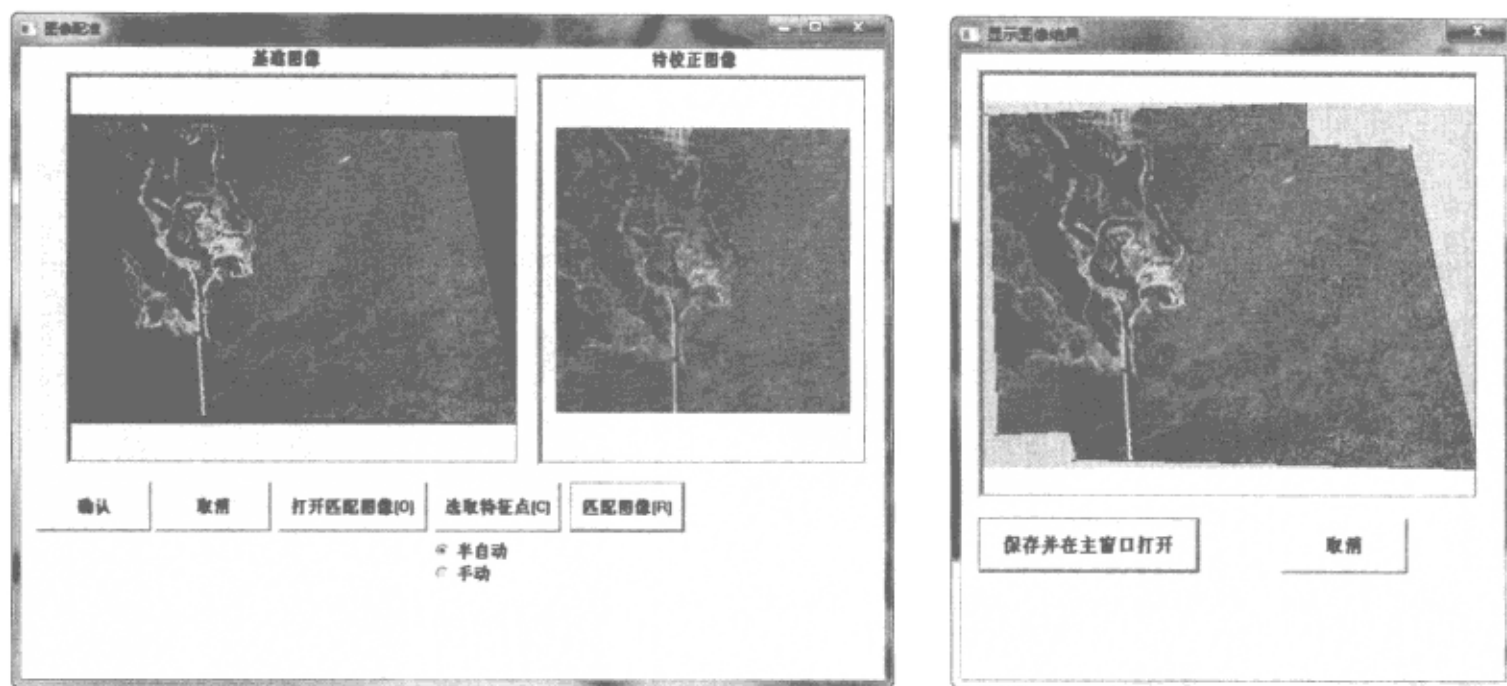


图 7-4 图像增强效果



a) 基准图像与待校正图像

b) 配准后的图像

图 7-5 图像配准效果

## 7.3 系统结构与流程

本章中的遥感图像配准系统主要由几何校正模块、图像增强模块和图像配准模块构成。本节主要介绍系统的总体结构和主要流程。

### 7.3.1 总体结构

遥感图像配准系统的总体结构如图 7-6 所示。

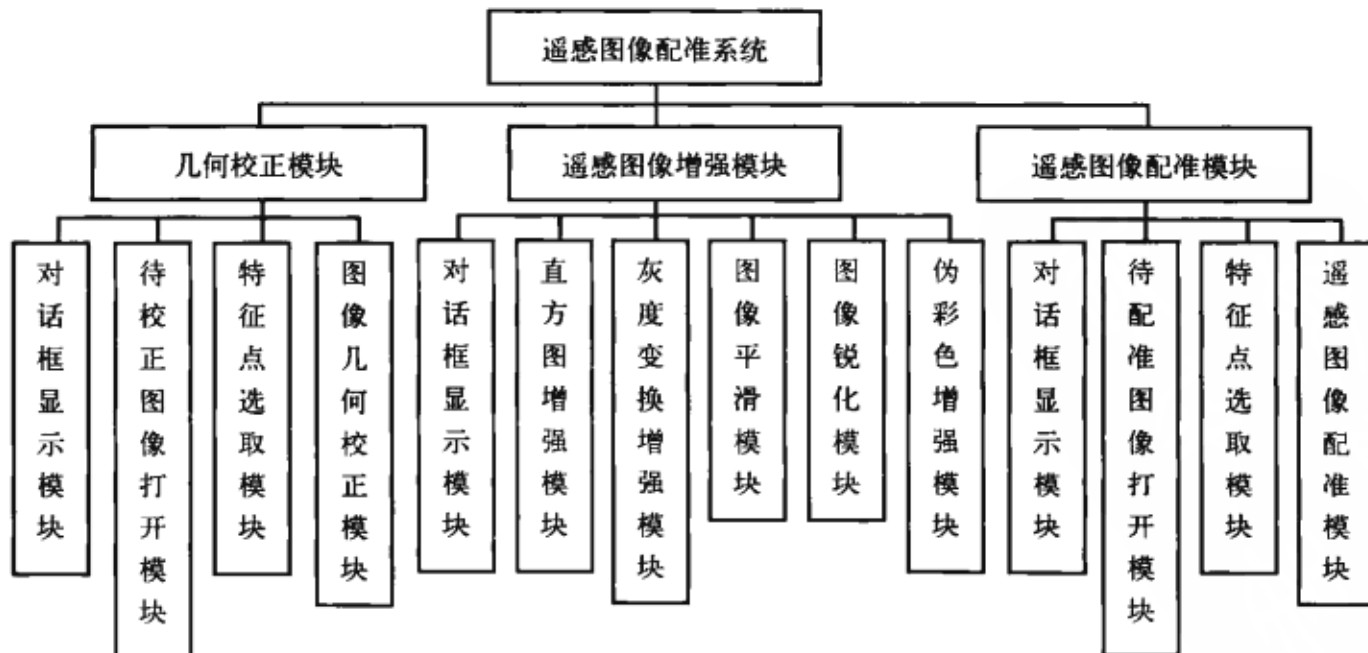


图 7-6 遥感图像配准系统总体结构

### 7.3.2 主要流程

遥感图像配准系统的主要流程如图 7-7 所示。

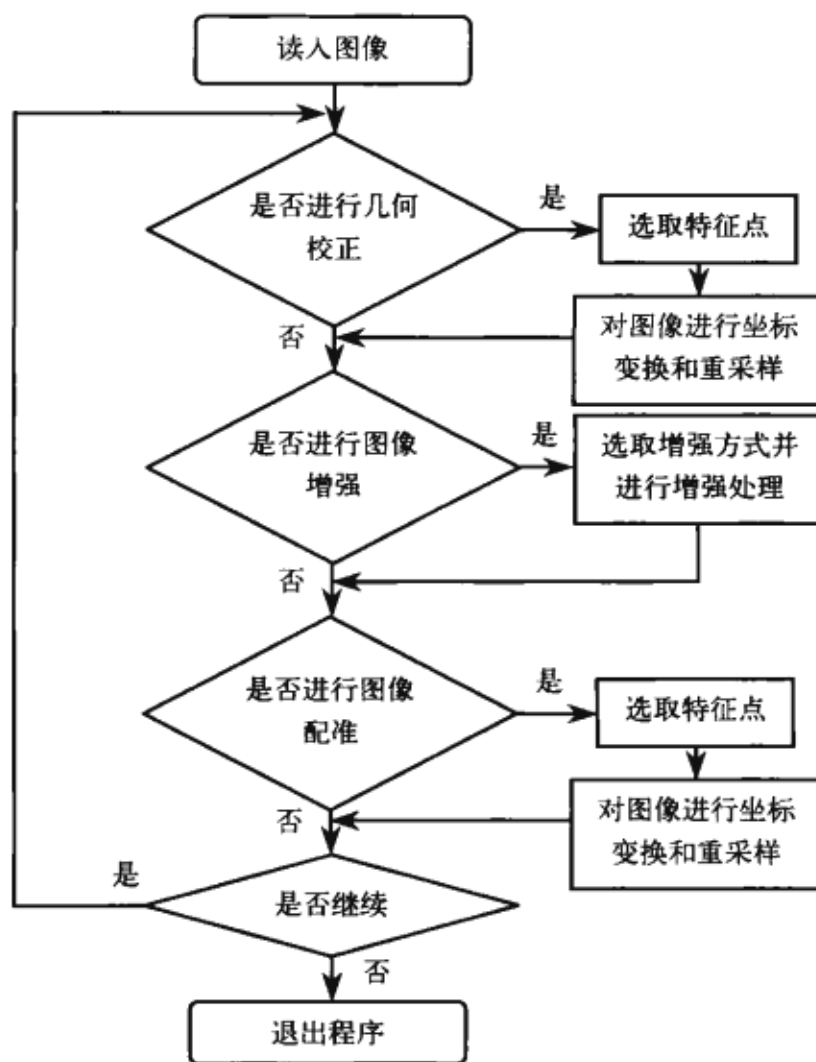


图 7-7 遥感图像配准系统主要流程图

## 7.4 编程实现

遥感图像配准系统采用 VC 2008 开发平台编程实现,自定义了用于处理 DIB 位图的 CDib 类。

### 7.4.1 CDib 类

DIB 是设备无关位图的缩写,它自带颜色信息,因此调色板管理非常容易。任何运行 Windows 操作系统的计算机都可以处理 DIB,它通常以 BMP 文件的形式保存。一个 BMP 文件大体上分成 4 个部分:位图文件头、位图信息头、调色板(颜色表)、DIB 图像数据。因此要对 BMP 格式的图像进行处理就要有一个 DIB 类,虽然 MFC 没有封装 DIB,但是在程序中使用 DIB 非常方便,

以下就是本系统自己构建的 DIB 类:

```
#define WIDTHBYTES(bits)    (((bits) + 31) / 32 * 4)
class CDib : public CObject
{
public:
    enum Alloc {noAlloc, crtAlloc, heapAlloc}; // 枚举类型, 表示内存分配的状况
    DECLARE_SERIAL(CDib)
public:
    LPVOID m_lpvColorTable;           // 调色板指针
    HBITMAP m_hBitmap;                // BITMAP 结构指针
    LPBYTE m_lpImage;                 // DIB 位图数据块地址
    LPBITMAPINFOHEADER m_lpBMPH;      // DIB 信息头指针
    HGLOBAL m_hGlobal;                // 全局的句柄, 用于内存映射文件中
    Alloc m_nBmpHAlloc;                // 表示信息头内存分配的状况
    Alloc m_nImageAlloc;              // 表示位图数据分配的状况
    DWORD m_dwSizeImage;               // DIB 位图中的字节数 (信息头和调色板数据除外)
    int m_nColorTableEntries;          // 调色板表项数
    HANDLE m_hFile;                    // 文件句柄
    HANDLE m_hMap;                     // 内存映射文件句柄
    LPVOID m_lpvFile;                  // 文件句柄
    HPALETTE m_hPalette;               // 调色板句柄
public:
    RGBQUAD GetPixel(int x, int y);    // 获取像素真实的颜色值
    LONG GetPixelOffset(int x, int y);  // 获取像素在图像数据块中的位置
    CSize GetDibSaveDim();              // 获取 DIB 位图数据块的存储尺寸
    BOOL IsEmpty();                     // 判断 DIB 是否为空
    WORD PaletteSize();                 // 计算调色板的表项数
    CDib();                             // 构造函数
    // 根据指定的位图尺寸和像素位数来构造 CDib 实例
    CDib(CSize size, int nBitCount);
    ~CDib();                             // 析构函数
    int GetSizeImage() {return m_dwSizeImage;} // 获取 DIB 位图中数据的字节数
    int GetSizeHeader()                 // 获取信息头和调色板的尺寸
    {return sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * m_nColorTableEntries;}
    CSize GetDimensions();               // 获取以像素表示的 DIB 的宽度和高度
    void Convert256toGray();              // 灰度化函数

    // 以读模式打开内存映射文件, 并将其与 CDib 对象进行关联
    BOOL AttachMapFile(const char* strPathname, BOOL bShare = FALSE);
    // 创建一个新的内存映射文件, 并进行数据的复制
    BOOL CopyToMapFile(const char* strPathname);
    // 用内存中的 DIB 与已有的 CDib 对象关联
    BOOL AttachMemory(LPVOID lpvMem, BOOL bMustDelete = FALSE, HGLOBAL hGlobal = NULL);
    // 将 CDib 对象按照指定的尺寸输出到显示器 (或者打印机)
    BOOL Draw(CDC* pDC, CPoint origin, CSize size);
};
```



## 数字图像处理典型案例详解

```

// 创建一个 DIB, 图像内存将不被初始化
HBITMAP CreateSection(CDC* pDC = NULL);
// 将 CDib 对象的逻辑调色板选入设备上下文, 然后实现该调色板
UINT UsePalette(CDC* pDC, BOOL bBackground = FALSE);
// 如果存在调色板, 那么读取它, 并创建一个 Windows 调色板
BOOL MakePalette();
// 将 DIB 重新生成压缩的或者不压缩的 DIB
BOOL Compress(CDC* pDC, BOOL bCompress = TRUE);
// 从以后的 DIB 中创建 DDB 位图, 实现 DIB 到 DDB 的转换
HBITMAP CreateBitmap(CDC* pDC);
// 从 DDB 中创建 DIB, 实现 DDB 到 DIB 的转换
BOOL ConvertFromDDB(HBITMAP hBitmap, HPALETTE hPal);
// 从文件中读取数据, 并填充文件头、信息头、调色板和位图数据
BOOL Read(CFile* pFile);
// 从 BMP 文件中读取信息头, 调用 CreateDIBSection 来分配位图数据内存, 然后将
// 位图从该文件读入到刚才分配的内存
BOOL ReadSection(CFile* pFile, CDC* pDC = NULL);
BOOL Write(CFile* pFile); // 将 DIB 从 CDib 对象写入文件
void Serialize(CArchive& ar); // 串行化过程
// 清空 DIB, 释放已经分配的内存, 并且关闭映射文件
void Empty();
void ComputePaletteSize(int nBitCount); // 计算调色板的尺寸
private:
void DetachMapFile(); // 断开映射文件的关联
void ComputeMetrics(); // 计算调色板和位图尺寸
};

```

通过以上这个类, 本系统就可以对 256 色 BMP 遥感图像进行读取和各种处理。值得注意的是, 为了能够对图像进行更好的操作, 本系统在读取图像后都会对其进行灰度化, 在上面这个类中 `Convert256toGray()` 就是灰度化函数, 它的代码如下:

```

void CDib::Convert256toGray()
{
    BYTE bMap[256]; // 灰度映射表
    LPRGBQUAD pDibQuad;
    LONG lWidth, lHeight;
    lWidth = m_lpBMPInfo->biWidth; // 获取图像宽度
    lHeight = m_lpBMPInfo->biHeight; // 获取图像高度

    // 计算灰度映射表 (保存各个颜色的灰度值), 并更新 DIB 调色板
    int i;
    for (i = 0; i < 256; i++)
    {
        pDibQuad = (LPRGBQUAD) (m_lpvColorTable + i);
        bMap[i] = (BYTE) (0.299 * pDibQuad->rgbRed + 0.587 * pDibQuad->rgbGreen +
            0.114 * pDibQuad->rgbBlue + 0.5);
    }
}

```

```

        pDibQuad->rgbRed = i;           // 更新 DIB 调色板红色分量
        pDibQuad->rgbGreen = i;         // 更新 DIB 调色板绿色分量
        pDibQuad->rgbBlue = i;          // 更新 DIB 调色板蓝色分量
        pDibQuad->rgbReserved = 0;      // 更新 DIB 调色板保留位
    }
    // 更换每个像素的颜色索引 (即按照灰度映射表转换成灰度值)
    for(int i=0;i<lHeight;i++)
        for(int j=0;j<lWidth;j++)
            m_lpImage[i*lWidth + j] = bMap[m_lpImage[i*lWidth + j]];
}

```

读取图片并进行灰度化后, 本系统就可以进入 3 大模块对图像进行相应操作了。

### 7.4.2 几何校正模块

在主窗口打开了遥感图像后, 用户就可以通过单击【几何校正】菜单打开【几何校正】对话框, 进入几何校正模块。

此模块使用如下标志位:

- BOOL m\_bCallmgLoc: 计算图像位置的标志位。FALSE 表示还没有计算图像位置。
- BOOL m\_bChoseFeature: 选取特征点标志位。FALSE 表示还没有选取。
- BOOL auto\_Feature: 特征点选取方式标志位。FALSE 表示手动选择特征点, 在几何校正模块中只有手动选取特征点方式, 因此这个标志位一直是 FALSE。
- BOOL handle\_baseFeaturechosen: 手动选取特征点时使用此标志位, FALSE 表示配准图像上的点已经选取完毕但基准图像没选取对应点, TRUE 表示基准图像上已经选好对应点。

此模块中有如下常用变量:

- CDib\* m\_pDibInit: 基准图像。
- CDib\* m\_pDibSamp: 待校正图像。
- CDib\* m\_pDibResult: 校正后的图像。
- CRect m\_rectInitImage: 基准图像显示区域。
- CRect m\_rectResltImage: 待校正图像显示区域。
- CPoint m\_pPointSampl[3]: 待校正图像的特征点位置。
- CPoint m\_pPointBase[3]: 基准图像的特征点位置。
- int m\_nChsFeatureNum: 选取的特征点个数。

#### 1. 对话框显示模块

几何校正对话框的显示主要是通过 CDlgCor::OnPaint()实现的。由于此系统对遥感图像大小并未进行规定, 所以当打开此对话框第一次进入 CDlgCor::OnPaint()的时候, CDlgCor::OnPaint()将会调用 CDlgCor::CallImageLocation 对对话框中控件的位置、大小, 以及显示图像的位置进行计算。

## 数字图像处理典型案例详解

默认的图像大小为  $352 \times 288$ ，如果图像小于此大小，则控件大小设置为  $352 \times 288$ ，并将图像放置在控件中间。具体代码如下：

```
void CDlgCor::OnPaint()
{
    CPaintDC dc(this); // 获取当前窗口的设备环境
    if(!m_bCalImgLoc)
        CalImageLocation(); // 如果还没有计算图像的位置，则进行计算
    CSize sizeDisplay;
    CPoint pointDisplay;
    // 显示基准图像
    if(!m_pDibInit->IsEmpty()){
        sizeDisplay.cx=m_pDibInit->m_lpBMP->biWidth;
        sizeDisplay.cy=m_pDibInit->m_lpBMP->biHeight;
        pointDisplay.x = m_rectInitImage.left;
        pointDisplay.y = m_rectInitImage.top;
        m_pDibInit->Draw(&dc,pointDisplay,sizeDisplay);
    }
    // 显示待校正图像
    if(!m_pDibSamp->IsEmpty()){
        sizeDisplay.cx=m_pDibSamp->m_lpBMP->biWidth;
        sizeDisplay.cy=m_pDibSamp->m_lpBMP->biHeight;
        pointDisplay.x = m_rectResltImage.left;
        pointDisplay.y = m_rectResltImage.top;
        m_pDibSamp->Draw(&dc,pointDisplay,sizeDisplay);
    }
    // 显示特征点与配准的特征点
    DrawFeature(&dc);
}
```

其中调用的 `CDlgCor::DrawFeature` 函数是根据类的成员变量确定特征点的数目和位置，并在图像中进行显示，代码如下：

```
void CDlgCor::DrawFeature(CDC* pDC)
{
    int i; // 循环变量
    CPoint pointTemp;
    int nRadius; // 半径
    nRadius = 5;
    pDC->SelectStockObject(HOLLOW_BRUSH); // 设置画图类型
    CPen penWhite(PS_SOLID,1,RGB(255,255,255)); // 声明画笔
    CPen *pOldPen;
    // 将画笔选入，并保存以前的画笔
    pOldPen = pDC->SelectObject(&penWhite);
    if(handle_baseFeaturechosen==FALSE&&m_bChoseFeature==TRUE)
```

```

        m_nChsFeatureNum++;
    for(i=0; i<m_nChsFeatureNum; i++)
    {
        // 首先显示特征点
        // 确定此点的显示位置
        pointTemp.x = m_pPointSampl[i].x + m_rectResltImage.left;
        pointTemp.y = m_pPointSampl[i].y + m_rectResltImage.top ;
        // 画出此特征点, 其中圆的半径为 nRadius
        CRect rectSamp(pointTemp.x-nRadius , pointTemp.y-nRadius ,
            pointTemp.x+nRadius , pointTemp.y+nRadius);
        pDC->Ellipse(rectSamp);
        // 再显示校正特征点
        // 确定此点的显示位置
        if(handle_baseFeaturechosen==TRUE||handle_baseFeaturechosen==FALSE&&
            i<m_nChsFeatureNum-1)
        {
            pointTemp.x = m_pPointBase[i].x + m_rectInitImage.left;
            pointTemp.y = m_pPointBase[i].y + m_rectInitImage.top ;
            // 画出此特征点, 其中圆的半径为 nRadius
            CRect rectBase(pointTemp.x-nRadius , pointTemp.y-nRadius ,
                pointTemp.x+nRadius , pointTemp.y+nRadius);
            pDC->Ellipse(rectBase);
        }
    }
    if(handle_baseFeaturechosen==FALSE&&m_bChoseFeature==TRUE)
        m_nChsFeatureNum--;
    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);
    penWhite.DeleteObject();
}

```

## 2. 特征点选取模块

当基准图片和待校正图片都在【几何校正】对话框中打开后, 用户就可以通过单击 **BREACK** 按钮进入 `CDlgCor::OnCorChoseFeature` 函数进行特征点的选取。该函数将设置选取特征点标志位, 然后弹出提示消息“请在待校正图像中选取特征点 (手动模式)”。其代码如下:

```

void CDlgCor::OnCorChoseFeature()
{
    // 如果待校正图像尚未打开, 则不能进行特征点选取工作
    if((m_pDibSamp->IsEmpty()))
    {
        AfxMessageBox("尚未打开待校正图像文件, 请打开待校正图像");
        return;
    }
    // 设置选取特征点标志位
    m_bChoseFeature = TRUE;
}

```

```
AfxMessageBox("请在待校正图像中选取特征点 (手动模式)");
}
```

接下来用户就可以按照消息提示在待校正图像中选取特征点了,当单击选取某点时,进入 CDlgCor::OnLButtonUp 函数,在该函数中判断当前的特征点选取模式,并进入相应的选取函数。其代码如下:

```
void CDlgCor::OnLButtonUp(UINT nFlags, CPoint point)
{
    if(auto_Feature==TRUE&&handle_baseFeaturechosen==TRUE)
        auto_Featurechosen(nFlags,point);
    else
        handle_Featurechosen(nFlags, point);
    // 更新显示
    Invalidate();
    CDialog::OnLButtonUp(nFlags, point);
}
```

由于几何校正只有手动选取特征点模式,因此本模块中只对手动模式进行分析,在遥感图像配准模块中再对半自动模式进行分析。

手动模式选取特征点需要先在待校正图像上选取一点,然后再在基准图像中选取一点,因此需要标志位 handle\_baseFeaturechosen 的配合,FALSE 时表示待校正图像上的点已经选取完毕,但基准图像没选取对应点,需要在基准图像上再选取一点,TRUE 表示基准图像上已经选好对应点,可以在待校正图像上进行下一点的选取。本系统最多可选取 3 个特征点。

进入手动模式选取函数后,系统需要进行以下几步来判断选中的点是否可用:

1) 计算特征点合法区域(比图像的区域要小一圈),共有待校正图像合法区域和基准图像合法区域两个区域。判断选取的点是否合法:如果 handle\_baseFeaturechosen 为 TRUE,则在待校正图像合法区域上选取的点合法;如果 handle\_baseFeaturechosen 为 FALSE,则在基准图像合法区域上选取的点不合法。

2) 判断选取的点是否是以前的特征点,如果是则将其去掉,并将对应的基准图像(或待校正图像)上的点也去掉。

3) 判断特征点是否已取够,如果特征点选取足够则弹出提示消息“已经选取了 3 个特征点,如果要继续选取,可以去掉校正不正确的特征点再进行选取”。

4) 此外还存在一种特殊情况:某点在待校正图片中处于合法区域,但是其同名点在基准图像中处于不合法区域(图像边缘)无法选取,那么此点也不是合法点,系统将此点及其对应的同名点取消。

如果经过以上判断此点可用,则将此点进行保存。

```
void CDlgCor::handle_Featurechosen(UINT nFlags, CPoint point)
```



```

{
    int i,j;           // 循环变量
    // 如果特征选取标志位为 TRUE, 则进行特征点的选取和校正, 否则退出
    if(!m_bChoseFeature)
        return;
    //待校正图像的特征选取区域, 在这里选择特征点的选择区域要比图像的区域小一圈
    CRect rectChosel;
    rectChosel.bottom = m_rectResltImage.bottom - 5;
    rectChosel.top     = m_rectResltImage.top + 5;
    rectChosel.left    = m_rectResltImage.left + 5;
    rectChosel.right   = m_rectResltImage.right - 5;
    // 基准图像的特征选取区域, 在这里选择特征点的选择区域要比图像的区域小一圈
    CRect rectChoose2;
    rectChoose2.bottom = m_rectInitImage.bottom - 5;
    rectChoose2.top     = m_rectInitImage.top + 5;
    rectChoose2.left    = m_rectInitImage.left + 5;
    rectChoose2.right   = m_rectInitImage.right - 5;

    CRect rectChoose3;
    rectChoose3.bottom = m_rectInitImage.bottom ;
    rectChoose3.top     = m_rectInitImage.top ;
    rectChoose3.left    = m_rectInitImage.left ;
    rectChoose3.right   = m_rectInitImage.right;
    // 特征点的区域
    CRect rectFeature1;
    CRect rectFeature2;
    // 标志位, 表示此点是否为已经选择的特征点
    BOOL bFlag1 = FALSE;
    BOOL bFlag2 = FALSE;
    if(handle_baseFeaturechosen==TRUE)
    {
        // 判断此点是否合法, 并判断此点是否已经选择, 如果是, 则去掉此点
        if(rectChosel.PtInRect(point))
        {
            // 如果所选择的特征点是以前的特征点, 则去掉此点
            for( i = 0; i<m_nChsFeatureNum; i++)
            {
                // 选择特征点的显示区域, 以便对特征点进行取舍
                rectFeature1.bottom = m_pPointSampl[i].y + m_rectResltImage.top + 5;
                rectFeature1.top     = m_pPointSampl[i].y + m_rectResltImage.top - 5;
                rectFeature1.left    = m_pPointSampl[i].x + m_rectResltImage.left - 5;
                rectFeature1.right   = m_pPointSampl[i].x + m_rectResltImage.left + 5;
                // 判断所选择的特征点是否为原来选择的特征点
                // 如果是, 则去掉此特征点
                if(rectFeature1.PtInRect(point))
                {
                    // 将后面的特征点向前移动一位, 去掉所选择的特征点

```



```

        for(j=i; j<m_nChsFeatureNum-1; j++)
        {
            m_pPointSampl[j] = m_pPointSampl[j+1];
            m_pPointBase[j] = m_pPointBase[j+1];
        }
        m_nChsFeatureNum--;           // 将特征点的计数减一
        Invalidate();                 // 更新显示
        bFlag1 = TRUE;                // 设置标志位
        return;                       // 退出
    }
}
// 判断特征点是否已选够
if(m_nChsFeatureNum == 3)
{
    AfxMessageBox("已经选取了 3 个特征点, 如果要继续选取, 可以去掉校正不正确的特征点再进行选取");
    return;
}
// 如果此点是需要选取的, 则进行相关操作
if(!bFlag1)
{
    // 将此特征点选取, 特征点的坐标是以图像的左上角为原点确定的
    m_pPointSampl[m_nChsFeatureNum].x = point.x - m_rectResltImage.left;
    m_pPointSampl[m_nChsFeatureNum].y = point.y - m_rectResltImage.top;
    handle_baseFeaturechosen=FALSE;
    this->UpdateData();
    this->Invalidate();
    AfxMessageBox("请在左侧基准图像中选取对应特征点");
}
}
else
{
    // 判断此点是否合法, 并判断此点是否已经选择, 如果是, 则去掉此点
    if(rectChoose3.PtInRect(point))
    {
        if(!rectChoose2.PtInRect(point))
        {
            AfxMessageBox("请在图像 5 像素以内选取特征点");
            handle_baseFeaturechosen=TRUE;
            return;
        }
        // 如果所选择的特征点是以前的特征点, 则去掉此点
        for( i = 0; i<m_nChsFeatureNum; i++)
        {
            // 选择特征点的显示区域, 以便对特征点进行取舍
            rectFeature2.bottom = m_pPointSampl[i].y + m_rectInitImage.top + 5;

```

```

rectFeature2.top    = m_pPointSampl[i].y + m_rectInitImage.top - 5;
rectFeature2.left   = m_pPointSampl[i].x + m_rectInitImage.left - 5;
rectFeature2.right  = m_pPointSampl[i].x + m_rectInitImage.left + 5;
// 判断所选择的特征点是否为原来选择的特征点
// 如果是, 则去掉此特征点
    if (rectFeature2.PtInRect(point))
    {
        // 将后面的特征点向前移动一位, 去掉所选择的特征点
        for (j=i; j<m_nChsFeatureNum-1; j++)
        {
            m_pPointSampl[j] = m_pPointSampl[j+1];
            m_pPointBase[j]  = m_pPointBase[j+1];
        }
        m_nChsFeatureNum--;           // 将特征点的计数减一
        Invalidate();                 // 更新显示
        bFlag2 = TRUE;                // 设置标志位
        return;
    }
}
// 如果此点是需要选取的, 则进行相关操作
if (!bFlag2)
{
    // 将此特征点选取, 注意特征点的坐标是以图像的左上角为原点确定的
    m_pPointBase[m_nChsFeatureNum].x = point.x - m_rectInitImage.left;
    m_pPointBase[m_nChsFeatureNum].y = point.y - m_rectInitImage.top;
    // 将特征点计数加一
    m_nChsFeatureNum++;
    handle_baseFeatureChosen=TRUE;
}
}
}
}

```

### 3. 图像几何校正模块

根据 7.1.1 节介绍的技术原理, 首先采用二元  $n$  次多项式对遥感图像进行坐标空间转换, 然后再用最近邻法对变换后的图像进行重采样, 从而得到校正后的遥感图像。

这里, 本系统取  $n=1$ , 即使用二元一次多项式对遥感图像进行坐标空间变换, 这样可以进行线性的坐标变换, 解决比例尺、中心移动、歪斜等方面的几何畸变。因为  $n=1$ , 所以二元一次多项式有 6 个系数, 本系统至少要取 3 个 GCP。

通过以上模块的前期工作, 本系统已经取得了 3 个特征点, 在本模块中, 系统将使用 CDlgCor::OnCorCor() 函数对遥感图像 m\_pDibSamp 进行几何校正。

该函数包括以下几步:

- 1) 确定已经选取特征点, 且至少需要选取 3 个以上。

- 2) 利用选取的特征点, 通过最小二乘法计算得到二元一次多项式系数。
  - 3) 得到系数后, 对图像进行空间变换, 最后通过最近邻法对图像进行重采样。
- 其代码如下:

```
void CDlgCor::OnCorCor()
{
    // 进行校正之前, 判断是否已经选取特征点
    if(!m_bChoseFeature)
    {
        AfxMessageBox("还没有选取特征点, 请先选取特征点");
        return;
    }
    // 如果选取的特征点数目不够, 也不能进行处理
    if(m_nChsFeatureNum < 3)
    {
        AfxMessageBox("请选择 3 个特征点, 再进行图像校正");
        return;
    }
    // 空间变换系数为 6 个。此系数为从基准图像到待校正图像的变换系数
    double *pDbBs2SpAffPara;
    pDbBs2SpAffPara = new double[2*3];
    // 空间变换的系数为 6 个。此系数为从待校正图像到基准图像的变换系数
    double *pDbSp2BsAffPara;
    pDbSp2BsAffPara = new double[2*3];

    // 计算从基准图像到待校正图像的空间变换系数
    GetAffinePara(m_pPointBase, m_pPointSampl, pDbSp2BsAffPara);
    // 计算从待校正图像到基准图像的空间变换系数
    GetAffinePara(m_pPointSampl, m_pPointBase, pDbBs2SpAffPara);
    // 计算图像经过变换后的尺寸大小
    CRect rectAftAff;
    rectAftAff = GetAftAffDim(pDbSp2BsAffPara);

    // 根据图像的尺寸大小创建新的图像
    int nNewImgWidth, nNewImgHeight;
    nNewImgWidth = rectAftAff.right - rectAftAff.left;
    nNewImgHeight = rectAftAff.bottom - rectAftAff.top;
    // 将校正后的图像放入新的图像中
    LPBYTE lpSampImg;
    lpSampImg = SetSampImgAftAff(pDbBs2SpAffPara, rectAftAff);
    // 将此图像用 CDib 类封装
    m_pDibResult = new CDib(CSize(nNewImgWidth, nNewImgHeight), 8);
    // 计算结果图像的存储大小尺寸
    CSize sizeSaveResult;
    sizeSaveResult = m_pDibResult->GetDibSaveDim();
    // 复制调色板
```

```

memcpy(m_pDibResult->m_lpvColorTable, m_pDibInit->m_lpvColorTable,
        m_pDibResult->m_nColorTableEntries*sizeof( RGBQUAD));
// 应用调色板
m_pDibResult->MakePalette();
// 分配内存给合并后的图像
LPBYTE lpImgResult;
lpImgResult = (LPBYTE)new unsigned char[sizeSaveResult.cx * sizeSaveResult.cy];

// 对图像进行赋值
for( int i=0; i<nNewImgWidth; i++)
    for( int j=0; j<nNewImgHeight; j++)
    {
        int nX = i;
        int nY = sizeSaveResult.cy - j - 1;
        lpImgResult[nY*sizeSaveResult.cx + nX] = lpSampImg[j*nNewImgWidth + i];
    }
// 将指针赋值给 CDib 类的数据
m_pDibResult->m_lpImage = lpImgResult;
// 显示校正后的图像
CDlgAftReg* pDlg;
pDlg = new CDlgAftReg(NULL, m_pDibResult);
pDlg->DoModal();

// 删除对象
delete pDlg;
// 释放已分配内存
delete[]lpSampImg;
delete[]pDbBs2SpAffPara;
delete[]pDbSp2BsAffPara;
}

```

在以上函数中，调用了 3 个函数，具体分析如下。

1) 首先调用 CDlgCor::GetAffinePara 函数，该函数的作用是根据得到的 3 对校正的特征点，计算空间变换系数，并将其存放在两个输入参数所指向的内存中。其代码如下：

```

void CDlgCor::GetAffinePara(CPoint* pPointBase, CPoint* pPointSampl, double*
pDbAffPara)
{
    // pDbBMatrix 中存放的是基准图像中特征点的坐标，
    // 大小为 2*m_nChsFeatureNum，其中前 m_nChsFeatureNum 为 X 坐标
    double *pDbBMatrix;
    pDbBMatrix = new double[2*m_nChsFeatureNum];

    // pDbSMatrix 中存放的是待校正图像中特征点的扩展坐标，
    // 大小为 3*m_nChsFeatureNum，其中前 m_nChsFeatureNum 为 X 坐标
    // 中间 m_nChsFeatureNum 为 Y 坐标，后面 m_nChsFeatureNum 为 1
}

```

```

double *pDbSMatrix;
pDbSMatrix = new double[3*m_nChsFeatureNum];

// pDbSMatrixT 中存放的 pDbSMatrix 的转置矩阵, 大小为 m_nChsFeatureNum*3
double *pDbSMatrixT;
pDbSMatrixT = new double[m_nChsFeatureNum*3];

// pDbInvMatrix 为临时变量, 存放的是 pDbSMatrix*pDbSMatrixT 的逆, 大小为 3*3
double *pDbInvMatrix;
pDbInvMatrix = new double[3*3];

// 临时内存
double *pDbTemp;
pDbTemp = new double[2*3];

int count;           // 循环变量
// 给矩阵赋值
for(count = 0; count<m_nChsFeatureNum; count++)
{
    pDbBMatrix[0*m_nChsFeatureNum + count] = pPointBase[count].x;
    pDbBMatrix[1*m_nChsFeatureNum + count] = pPointBase[count].y;
    pDbSMatrix[0*m_nChsFeatureNum + count] = pPointSampl[count].x;
    pDbSMatrix[1*m_nChsFeatureNum + count] = pPointSampl[count].y;
    pDbSMatrix[2*m_nChsFeatureNum + count] = 1;
    pDbSMatrixT[count*m_nChsFeatureNum + 0] = pPointSampl[count].x;
    pDbSMatrixT[count*m_nChsFeatureNum + 1] = pPointSampl[count].y;
    pDbSMatrixT[count*m_nChsFeatureNum + 2] = 1;
}
// 计算 pDbSMatrix*pDbSMatrixT, 并将结果放入 pDbInvMatrix 中
CalMatProduct(pDbSMatrix, pDbSMatrixT, pDbInvMatrix, 3, 3, m_nChsFeatureNum);
// 计算 pDbInvMatrix 的逆矩阵
CalInvMatrix(pDbInvMatrix, 3);
// 计算空间变换系数
CalMatProduct(pDbBMatrix, pDbSMatrixT, pDbTemp, 2, 3, m_nChsFeatureNum);
CalMatProduct(pDbTemp, pDbInvMatrix, pDbAffPara, 2, 3, 3);

// 释放内存
delete[]pDbBMatrix;
delete[]pDbSMatrix;
delete[]pDbSMatrixT;
delete[]pDbInvMatrix;
delete[]pDbTemp;
}

```

上述代码调用了 CDlgCor::CalMatProduct(double\* pDbSrc1, double \*pDbSrc2, double \*pDbDest, int y, int x, int z)函数和 CDlgCor::CalInvMatrix(double \*pDbSrc, int nLen)函数,前者计算两个矩阵的

相乘,然后将相乘的结果存放在 pDbDest 中,其中 pDbSrc1 的大小为  $nX \times nZ$ , pDbSrc2 的大小为  $nZ \times nY$ , pDbDest 的大小为  $nX \times nY$ ; 后者计算矩阵 pDbSrc 的逆矩阵,其中 pDbSrc 的大小为  $nLen \times nLen$ 。

2) 然后调用 CRect CDlgCor::GetAftAffDim(double\* pDbAffPara)函数,该函数输入为 pDbAffPara 是空间变换系数矩阵,返回值为待校正图像经空间变换后的区域。

3) 最后通过 CDlgCor::SetSampImgAftAff()函数得到校正后的图像,此函数的输入参数有 pDbAffPara (仿射变换系数矩阵)和 rectNewImg (变换后图像的大小尺寸),返回变换后的图像。此函数根据空间变换系数,对待校正图像进行空间变换,最后调用 CDlgCor::CalSpline 进行最近邻插值得到校正后的图像,并返回此图像指针,图像的大小为 rectNewImg,其代码如下:

```
LPBYTE CDlgCor::SetSampImgAftAff(double* pDbAffPara, CRect rectNewImg)
{
    // pUnchSect 是 4*4 大小的矩阵数组
    unsigned char *pUnchSect;
    pUnchSect = new unsigned char[4*4];
    // 新的图像宽度和高度
    int nNewImgWidth, nNewImgHeight;
    nNewImgWidth = rectNewImg.right - rectNewImg.left;
    nNewImgHeight = rectNewImg.bottom - rectNewImg.top;
    // 待校正图像的宽度和高度
    int nSampImgWidth, nSampImgHeight;
    nSampImgWidth = m_pDibSamp->m_lpBMPInfo->biWidth;
    nSampImgHeight = m_pDibSamp->m_lpBMPInfo->biHeight;
    // 待校正图像的存储宽度
    int nSampSaveWidth;
    nSampSaveWidth = m_pDibSamp->GetDibSaveDim().cx;

    // pUnchAftAffSamp 是一个大小为 rectNewImg 的图像,
    // 其中 rectNewImg 表示变换后的图像大小
    unsigned char *pUnchAftAffSamp;
    pUnchAftAffSamp = new unsigned char[nNewImgWidth * nNewImgHeight];

    double tx, ty;
    // 计算在变换后的图像的数据
    for(int i=0; i<rectNewImg.bottom-rectNewImg.top; i++)
        for(int j=0; j<rectNewImg.right-rectNewImg.left; j++)
        {
            tx = pDbAffPara[0*3 + 0]*(j+rectNewImg.left) +
                pDbAffPara[0*3 + 1]*(i+rectNewImg.top) + pDbAffPara[0*3 + 2];
            ty = pDbAffPara[1*3 + 0]*(j+rectNewImg.left) +
                pDbAffPara[1*3 + 1]*(i+rectNewImg.top) + pDbAffPara[1*3 + 2];

            for(int m=(int)ty-1; m<=(int)ty+2; m++)
```



```

        for(int n=(int)tx-1;n<=(int)tx+2;n++)
        {
            if(m<0||m>=nSamplImgHeight||n<0||n>=nSamplImgWidth)
                pUnchSect[(m-(int)ty+1)*4 + (n-(int)tx+1)] = 0;
            else
                pUnchSect[(m-(int)ty+1)*4 + (n-(int)tx+1)] =
                m_pDibSamp->m_lpImage[(nSamplImgHeight-m-1)*
                nSampSaveWidth + n];
        }
        // 确定变换的坐标
        ty = ty - (int)ty + 1;
        tx = tx - (int)tx + 1;
        // 确定变换后此坐标的数值
        pUnchAftAffSamp[i*nNewImgWidth + j] = CalSpline(pUnchSect,tx,ty);
    }
    // 释放内存
    delete[]pUnchSect;
    // 返回指针
    return (LPBYTE)pUnchAftAffSamp;
}

```

其中通过最近邻法进行重采样的函数 CDlgCor::CalSpline 代码如下:

```

unsigned char CDlgCor::CalSpline(unsigned char *pUnchCorr, double x, double y)
{
    double ret=0, Cx, Cy;
    double Temp;
    for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
            Temp = pUnchCorr[i*4 + j];
            if(fabs(y-i)<1)
                Cy = 1-2*fabs(y-i)*fabs(y-i)+fabs(y-i)*fabs(y-i)*fabs(y-i);
            if(fabs(y-i)>=1)
                Cy = 4-8*fabs(y-i)+5*fabs(y-i)*fabs(y-i)-fabs(y-i)*fabs(y-i)*fabs(y-i);
            if(fabs(x-j)<1)
                Cx = 1-2*fabs(x-j)*fabs(x-j)+fabs(x-j)*fabs(x-j)*fabs(x-j);
            if(fabs(x-j)>=1)
                Cx = 4-8*fabs(x-j)+5*fabs(x-j)*fabs(x-j)-fabs(x-j)*fabs(x-j)*fabs(x-j);
            ret += Temp*Cy*Cx;
        }
    if(ret<0)
        ret=0;
    if(ret>255)
        ret=255;
    return (unsigned char)ret;
}

```

### 7.4.3 遥感图像增强模块

在主窗口中打开了遥感图像后，用户就可以通过单击【图像增强】菜单打开【图像增强】对话框，进入遥感图像增强模块。

在此模块中使用 `BOOL m_bCallImgLoc` 标志位标记计算图像位置。`FALSE` 表示还没有计算图像位置。

此模块中有如下常用变量：

- `CDib* m_pDibInit`：基准图像。
- `CDib* m_pDibResult`：增强后的图像。
- `CRect m_rectInitImage`：基准图像显示区域。
- `CRect m_rectResltImage`：增强后图像显示区域。

#### 1. 对话框显示模块

在此模块中同样使用 `CDlgEnhance::OnPaint()` 显示图像增强对话框，并调用 `CDlgEnhance::CallImageLocation()` 对对话框中控件的位置、大小，以及显示图像的位置进行计算。

同时，由于本系统中提供的图像增强方法很多，为了方便地进行选择，本对话框使用了大量的控件，相关控件列表如表 7-1 所示。

表 7-1 图像增强对话框中增强方式相关控件

ID	CAPTION	调用函数
<code>IDC_BUTTON_HISTOGRAM</code>	直方图增强	<code>OnBnClickedButtonHistogram</code>
<code>IDC_HISTOGRAM_EQUAL</code>	均衡化	<code>OnBnClickedHistogramEqual</code>
<code>IDC_HISTOGRAM_MATCH</code>	规定化	<code>OnBnClickedHistogramMatch</code>
<code>IDC_BUTTON_LINEAR</code>	灰度增强	<code>OnBnClickedButtonLinear</code>
<code>IDC_LINEAR_1</code>	线性灰度	<code>OnBnClickedLinear1</code>
<code>IDC_LINEAR_2</code>	分段线性	<code>OnBnClickedLinear2</code>
<code>IDC_LINEAR_3</code>	对数非线性	<code>OnBnClickedLinear3</code>
<code>IDC_BUTTON_SMOOTH</code>	平滑处理	<code>OnBnClickedButtonSmooth</code>
<code>IDC_SMOOTH_1</code>	邻域平均	<code>OnBnClickedSmooth1</code>
<code>IDC_SMOOTH_2</code>	加权平均	<code>OnBnClickedSmooth2</code>
<code>IDC_SMOOTH_3</code>	选择式掩膜	<code>OnBnClickedSmooth3</code>
<code>IDC_SMOOTH_4</code>	中值滤波	<code>OnBnClickedSmooth4</code>
<code>IDC_BUTTON_SHARPEN</code>	锐化处理	<code>OnBnClickedButtonSharpen</code>
<code>IDC_SHARPEN_1</code>	门限梯度	<code>OnBnClickedSharpen1</code>
<code>IDC_SHARPEN_2</code>	拉普拉斯	<code>OnBnClickedSharpen2</code>
<code>IDC_BUTTON_COLOR</code>	伪彩色增强	<code>OnBnClickedButtonColor</code>

## 2. 直方图增强模块

直方图增强模块中主要有两种方法：直方图均衡化和直方图规定化。

### (1) 直方图统计函数

在实现两种方法前，首先需要有一个直方图统计函数，以便在直方图增强时调用。其实现代码如下：

```
void CDlgEnhance::Histogram_Statistic( float *probability)
{
    LPBYTE  lpSrc;                // 指向原图像的指针
    long i,j;                    // 循环变量
    int gray[256];               // 灰度计数
    BYTE pixel;                 // 像素值
    unsigned char* lpDIBBits;    // 基准图像数据指针
    lpDIBBits = (unsigned char *)m_pDibInit->m_lpImage;
    LONG lWidth=m_pDibInit->m_lpBMPInfo->biWidth;    // 获得原图像的宽度
    LONG lHeight=m_pDibInit->m_lpBMPInfo->biHeight;  // 获得原图像的高度
    LONG lLineBytes=WIDTHBYTES(lWidth*8);
    memset(gray,0,sizeof(gray));    // 灰度计数变量初始化
    //逐个扫描图像中的像素点，进行灰度计数统计
    for(j = 0; j <lHeight; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            // 指向原图像倒数第 j 行，第 i 个像素的指针
            lpSrc = (LPBYTE)lpDIBBits + lLineBytes * j + i;
            //取得当前指针处的像素值，注意要转换为 BYTE 类型
            pixel = (BYTE)*lpSrc;
            // 灰度统计计数
            gray[pixel]++;
        }
    }
    // 计算灰度概率密度
    for(i=0;i<256;i++)
        probability[i] = gray[i] / (lHeight * lWidth *1.0f);
}
```

### (2) 直方图均衡化

直方图均衡化的代码如下：

```
BOOL CDlgEnhance::Histogram_Equalization()
{
    unsigned char*  lpSrc;        // 原图像数据指针
    long i,j;                // 循环变量
    float fPro[256];          // 原图像灰度分布概率密度变量
    float temp[256];          // 中间变量
```

```

BYTE nRst[256];

lpSrc = (unsigned char *)m_pDibInit->m_lpImage;
LONG lWidth=m_pDibInit->m_lpBMiH->biWidth;    // 获得原图像的宽度
LONG lHeight=m_pDibInit->m_lpBMiH->biHeight;  // 获得原图像的高度
LONG lLineBytes=WIDTHBYTES(lWidth*8);
memset(temp, 0, sizeof(temp));                // 初始化中间变量 temp
Histogram_Statistic(fPro);                    // 获取原图像灰度分布的概率密度
//进行直方图均衡化处理
for(i = 0; i < 256; i++)
{
    if(i == 0)
        temp[0] = fPro[0];
    else
        temp[i] = temp[i-1] + fPro[i];
    nRst[i] = (BYTE)(255.0f * temp[i] + 0.5f);
}
m_pDibResult = new CDib(CSize(lWidth,lHeight), 8);
// 计算结果图像的存储大小尺寸
CSize sizeSaveResult;
sizeSaveResult = m_pDibResult->GetDibSaveDim();
// 复制调色板
memcpy(m_pDibResult->m_lpvColorTable, m_pDibInit->m_lpvColorTable,
        m_pDibResult->m_nColorTableEntries*sizeof(RGBQUAD));
// 应用调色板
m_pDibResult->MakePalette();

// 分配内存给合并后的图像
LPBYTE lpDst;
lpDst = (LPBYTE)new unsigned char[sizeSaveResult.cx * sizeSaveResult.cy];

//将直方图均衡化后的结果写到目标图像中
for(j = 0; j < lHeight; j++)
    for(i = 0; i < lWidth; i++)
        lpDst[lLineBytes * j + i] = (nRst[lpSrc[lLineBytes * j + i]]);
// 将指针赋值给 CDib 类的数据
m_pDibResult->m_lpImage = lpDst;
// 更新显示
this->UpdateData();
this->Invalidate();
return TRUE;
}

```

### (3) 直方图规定化

CDlgEnhance::Histogram\_Match()函数可对遥感图像进行直方图规定化,其代码如下:

```

BOOL CDlgEnhance::Histogram_Match()

```

```

{
    unsigned char*  lpSrc;    // 原图像数据指针
    long i,j;                // 循环变量
    float fPro[256];         // 原图像灰度分布概率密度变量
    float temp[256];         // 中间变量
    BYTE nMap[256];          // 灰度映射表变量
    BYTE bGray=64;
    int npMap[64];            // 规定直方图映射关系, 这里规定直方图的灰度级为 64
    float fpPro[64];          // 规定灰度分布概率
    float a=1.0f/(32.0f*63.0f);
    for(int i=0;i<64;i++)
    {
        npMap[i]=i*4;
        fpPro[i]=a*i;
    }
    lpSrc = (unsigned char *)m_pDibInit->m_lpImage;
    LONG lWidth=m_pDibInit->m_lpBMIT->biWidth;    // 获得原图像的宽度
    LONG lHeight=m_pDibInit->m_lpBMIT->biHeight;    // 获得原图像的高度
    LONG lLineBytes=WIDTHBYTES(lWidth*8);
    // 初始化中间变量 temp
    memset(temp, 0, sizeof(temp));
    // 获取原图像灰度分布的概率密度
    Histogram_Statistic(fPro);
    // 计算原图像累计直方图
    for (i = 0; i < 256; i++)
    {
        if (i == 0)
            temp[0] = fPro[0];
        else
            temp[i] = temp[i-1] + fPro[i];
        fPro[i] = temp[i];
    }
    // 计算规定变换后的累计直方图
    for (i = 0; i < bGray; i++)
    {
        if (i == 0)
            temp[0] = fpPro[0];
        else
            temp[i] = temp[i-1] + fpPro[i];
        fpPro[i] = temp[i];
    }
    // 确定映射关系
    for (i = 0; i < 256; i++)
    {
        int m = 0;
        float min_value = 1.0f;
        // 枚举规定直方图各灰度
        // 最接近的规定直方图灰度级
        // 最小差值
    }
}

```

```

    for (j = 0; j < bGray; j++)
    {
        float now_value = 0.0f;           // 当前差值
        if (fPro[i] - fpPro[j] >= 0.0f)    // 差值计算
            now_value = fPro[i] - fpPro[j];
        else
            now_value = fpPro[j] - fPro[i];
        // 寻找最接近的规定直方图灰度级
        if (now_value < min_value)
        {
            m = j;                         // 最接近的灰度级
            min_value = now_value;         // 最小差值
        }
    }
    nMap[i] = npMap[m];                   // 建立灰度映射表
}
m_pDibResult = new CDib(CSize(lWidth, lHeight), 8);

// 计算结果图像的存储大小尺寸
CSize sizeSaveResult;
sizeSaveResult = m_pDibResult->GetDibSaveDim();
// 复制调色板
memcpy(m_pDibResult->m_lpvColorTable, m_pDibInit->m_lpvColorTable,
        m_pDibResult->m_nColorTableEntries*sizeof(RGBQUAD));
// 应用调色板
m_pDibResult->MakePalette();
// 分配内存给合并后的图像
LPBYTE lpDst;
lpDst = (LPBYTE)new unsigned char[sizeSaveResult.cx * sizeSaveResult.cy];

// 对各像素进行直方图规定化映射处理
for (j = 0; j < lHeight; j++)
    for (i = 0; i < lWidth; i++)
        lpDst[lLineBytes * j + i] = (nMap[lpSrc[lLineBytes * j + i]]);
// 将指针赋值给 CDib 类的数据
m_pDibResult->m_lpImage = lpDst;
// 更新显示
this->UpdateData();
this->Invalidate();
return TRUE;
}

```

由以上内容可以看出，两种增强函数处理的都是相同图像，只是核心原理不同。因此为了节省篇幅不再赘述，以下每种增强方法只展示其核心原理和核心代码。其中 lpSrc 用来表示原图像指针，lpDst 表示增强后的图像指针。



### 3. 灰度变换增强模块

本系统使用线性灰度增强、分段线性灰度增强和非线性灰度增强 3 种灰度变换增强方法。

#### (1) 线性灰度增强

下面给出遥感图像的灰度范围由[0, 255]压缩到[gMin, gMax]的核心代码:

```
long i, j;           // 循环变量
BYTE pixel;         // 像素值
BYTE gMin, gMax;     // 设置变换后的灰度区间
gMin=100;
gMax=200;
// 逐个扫描图像中的像素点, 进行灰度线性变换
for(j = 0; j < lHeight; j++)
{
    for(i = 0; i < lWidth; i++)
    {
        // 取得当前指针处的像素值, 注意要转换为 BYTE 类型
        pixel = (BYTE)lpSrc[lLineBytes * j + i];
        // 根据公式求出目标图像中与当前点对应的像素点的灰度值
        lpDst[lLineBytes * j + i] = (BYTE)((float)(gMax-gMin)/255)*pixel+gMin+0.5;
    }
}
```

#### (2) 分段线性灰度增强

分段线性灰度增强的核心代码如下:

```
BYTE gSrc1, gSrc2, gDst1, gDst2; // 设置分段点
gSrc1=100;
gSrc2=150;
gDst1=50;
gDst2=200;
BYTE pixel;           // 像素值
// 逐个扫描图像中的像素点, 进行灰度分段线性变换
for(j = 0; j < lHeight; j++)
{
    for(i = 0; i < lWidth; i++)
    {
        // 取得当前指针处的像素值, 注意要转换为 BYTE 类型
        pixel = (BYTE)lpSrc[lLineBytes * j + i];
        // 求出目标图像中与当前点对应的像素点的灰度值
        if(pixel < gSrc1)
            lpDst[lLineBytes * j + i] = (BYTE)((float)gDst1/gSrc1)*pixel+0.5;
        // 求出目标图像中与当前点对应的像素点的灰度值
        if((pixel >= gSrc1) && (pixel <= gSrc2))
            lpDst[lLineBytes * j + i] = (BYTE)((float)(gDst2-gDst1)/
                (gSrc2-gSrc1))*(pixel-gSrc1)+gDst1+0.5;
    }
}
```

```
//根据公式求出目标图像中与当前点对应的像素点的灰度值
if((pixel>gSrc2)&&(pixel<=255))
    lpDst[lLineBytes * j + i] = (BYTE)((float)(255-gDst2)/
        (255-gSrc2))*(pixel-gSrc2)+gDst2+0.5);
}
```

### (3) 非线性灰度增强

通过调整参数  $a$ 、 $b$ 、 $c$  可以调整非线性灰度增强的结果，核心代码如下：

```
BYTE pixel;           // 像素值
double a,b,c;         // 对数函数变换参数设置
a=50.0;
b=0.8;
c=1.05;
// 逐个扫描图像中的像素点，进行对数函数非线性灰度变换
for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth; i++)
    {
        // 取得当前指针处的像素值，注意要转换为 BYTE 类型
        pixel = (BYTE)lpSrc[lLineBytes * j + i];
        // 根据公式求出目标图像中与当前点对应的像素点的灰度值
        lpDst[lLineBytes * j + i] = (BYTE)((log((double)(pixel+1)))/(b*log(c))+a+0.5);
    }
}
```

## 4. 图像平滑模块

图像平滑模块中包括邻域平均法、加权平均法、选择式掩膜法和中值滤波法。具体介绍如下。

### (1) 邻域平均法

下面给出遥感图像使用  $3 \times 3$  的模板  $\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$  的核心代码：

```
BYTE average;         // 领域均值变量
// 逐个扫描图像中的像素点，求其邻域均值
for(j = 1; j <lHeight-1; j++)
{
    for(i = 1;i <lWidth-1; i++)
    {
        // 求当前点及其周围 8 个点的均值
        average=(BYTE)((float)(lpSrc[(j-1)*lLineBytes+(i-1)]+
            lpSrc[(j-1)*lLineBytes+i]+lpSrc[(j-1)*lLineBytes+(i+1)]
```

```

+lpSrc[j*lLineBytes+(i-1)]+lpSrc[j*lLineBytes+i]+
lpSrc[j*lLineBytes+i+1]+lpSrc[(j+1)*lLineBytes+(i-1)]+
lpSrc[(j+1)*lLineBytes+i]+lpSrc[(j+1)*lLineBytes+i+1])/9+0.5);
// 将求得的均值赋值给目标图像中与当前点对应的像素点
lpDst[lLineBytes * j + i] = average;
    }
}

```

## (2) 加权平均法

下面给出遥感图像使用加权平均模板  $\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$  的核心代码:

```

int sum=0; // 模板中各个元素总和
BYTE value_average; // 领域加权均值变量
int Structure[3][3]={1,2,1,2,4,2,1,2,1}; // 定义加权模板
// 求模板中各元素权值总和
for (m = 0;m < 3;m++)
    for (n = 0;n < 3;n++)
        sum+=Structure[m][n];
// 逐个扫描图像中的像素点, 求其邻域加权均值
for(j = 1; j < lHeight-1; j++)
{
    for(i = 1;i < lWidth-1; i++)
    {
        //求加权均值
        value_average=(BYTE)((float)(lpSrc[(j+1)*lLineBytes+(i-1)]
            *Structure[0][0]+
            lpSrc[(j+1)*lLineBytes+i]*Structure[0][1]+
            lpSrc[(j+1)*lLineBytes+i+1]*Structure[0][2]+
            lpSrc[j*lLineBytes+(i-1)]*Structure[1][0] +
            lpSrc[j*lLineBytes+i]*Structure[1][1]+
            lpSrc[j*lLineBytes+i+1]*Structure[1][2]+
            lpSrc[(j-1)*lLineBytes+(i-1)]*Structure[2][0]+
            lpSrc[(j-1)*lLineBytes+i]*Structure[2][1]+
            lpSrc[(j-1)*lLineBytes+(i+1)]*Structure[2][2])/sum+0.5);
        //将求得的加权均值赋值给目标图像中与当前点对应的像素点
        lpDst[lLineBytes * j + i] = value_average;
    }
}

```

## (3) 选择式掩膜法

下面给出使用如图 7-8 所示的 5×5 模板窗口的核心代码。

0 0 0 0 0	0 0 0 0 0	0 1 1 1 0
0 1 1 1 0	1 1 0 0 0	0 1 1 1 0
0 1 1 1 0	1 1 1 0 0	0 0 1 0 0
0 1 1 1 0	1 1 0 0 0	0 0 0 0 0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0
a)	b)	c)
0 0 0 0 0	0 0 0 0 0	1 1 0 0 0
0 0 0 1 1	0 0 0 0 0	1 1 1 0 0
0 0 1 1 1	0 0 1 0 0	0 1 1 0 0
0 0 0 1 1	0 1 1 1 0	0 0 0 0 0
0 0 0 0 0	0 1 1 1 0	0 0 0 0 0
d)	e)	f)
0 0 0 1 1	0 0 0 0 0	0 0 0 0 0
0 0 1 1 1	0 0 0 0 0	0 0 0 0 0
0 0 1 1 0	0 0 1 1 0	0 1 1 0 0
0 0 0 0 0	0 0 1 1 1	1 1 1 0 0
0 0 0 0 0	0 0 0 1 1	1 1 0 0 0
g)	h)	i)

图 7-8 9种屏蔽窗口的模板

```

BYTE pixel[9];
float mean[9], var[9], varMin;
int nMin;
//求9种邻域的均值与方差
for(j=2; j<=lHeight-3; j++)
{
    for(i=2; i<=lWidth-3; i++)
    {
        //第1邻域
        pixel[0]=lpSrc[(j-1)*lLineBytes+(i-1)];
        pixel[1]=lpSrc[(j-1)*lLineBytes+i];
        pixel[2]=lpSrc[(j-1)*lLineBytes+(i+1)];
        pixel[3]=lpSrc[j*lLineBytes+(i-1)];
        pixel[4]=lpSrc[j*lLineBytes+i];
        pixel[5]=lpSrc[j*lLineBytes+(i+1)];
        pixel[6]=lpSrc[(j+1)*lLineBytes+(i-1)];
        pixel[7]=lpSrc[(j+1)*lLineBytes+i];
        pixel[8]=lpSrc[(j+1)*lLineBytes+(i+1)];
        mean[0]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
            pixel[4]+pixel[5]+pixel[6]+pixel[7]+pixel[8])/9;
        var[0]=0;
        for(n=0; n<=8; n++)
            var[0]+=pixel[n]*pixel[n]-mean[0]*mean[0];
    }
}

```

//领域各点的像素值  
//邻域均值, 邻域方差, 方差最小值  
//方差最小时的邻域号

```

//第2邻域
pixel[0]=lpSrc[(j-2)*lLineBytes+(i-1)];
pixel[1]=lpSrc[(j-2)*lLineBytes+i];
pixel[2]=lpSrc[(j-2)*lLineBytes+(i+1)];
pixel[3]=lpSrc[(j-1)*lLineBytes+(i-1)];
pixel[4]=lpSrc[(j-1)*lLineBytes+i];
pixel[5]=lpSrc[(j-1)*lLineBytes+(i+1)];
pixel[6]=lpSrc[j*lLineBytes+i];
mean[1]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
pixel[4]+pixel[5]+pixel[6])/7;
var[1]=0;
for(n=0;n<=6;n++)
    var[1]+=pixel[n]*pixel[n]-mean[1]*mean[1];

//第3邻域
pixel[0]=lpSrc[(j-1)*lLineBytes+(i-2)];
pixel[1]=lpSrc[(j-1)*lLineBytes+(i-1)];
pixel[2]=lpSrc[j*lLineBytes+(i-2)];
pixel[3]=lpSrc[j*lLineBytes+(i-1)];
pixel[4]=lpSrc[j*lLineBytes+i];
pixel[5]=lpSrc[(j+1)*lLineBytes+(i-2)];
pixel[6]=lpSrc[(j+1)*lLineBytes+(i-1)];
mean[2]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
pixel[4]+pixel[5]+pixel[6])/7;
var[2]=0;
for(n=0;n<=6;n++)
    var[2]+=pixel[n]*pixel[n]-mean[2]*mean[2];

//第4邻域
pixel[0]=lpSrc[j*lLineBytes+i];
pixel[1]=lpSrc[(j+1)*lLineBytes+(i-1)];
pixel[2]=lpSrc[(j+1)*lLineBytes+i];
pixel[3]=lpSrc[(j+1)*lLineBytes+(i+1)];
pixel[4]=lpSrc[(j+2)*lLineBytes+(i-1)];
pixel[5]=lpSrc[(j+2)*lLineBytes+i];
pixel[6]=lpSrc[(j+2)*lLineBytes+(i+1)];
mean[3]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
pixel[4]+pixel[5]+pixel[6])/7;
var[3]=0;
for(n=0;n<=6;n++)
    var[3]+=pixel[n]*pixel[n]-mean[3]*mean[3];

//第5邻域
pixel[0]=lpSrc[(j-1)*lLineBytes+(i+1)];
pixel[1]=lpSrc[(j-1)*lLineBytes+(i+2)];
pixel[2]=lpSrc[j*lLineBytes+i];

```

```

    pixel[3]=lpSrc[j*1LineBytes+(i+1)];
    pixel[4]=lpSrc[j*1LineBytes+(i+2)];
    pixel[5]=lpSrc[(j+1)*1LineBytes+(i+1)];
    pixel[6]=lpSrc[(j+1)*1LineBytes+(i+2)];
    mean[4]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
        pixel[4]+pixel[5]+pixel[6])/7;
    var[4]=0;
    for(n=0;n<=6;n++)
        var[4]+=pixel[n]*pixel[n]-mean[4]*mean[4];

    //第6邻域
    pixel[0]=lpSrc[(j-2)*1LineBytes+(i+1)];
    pixel[1]=lpSrc[(j-2)*1LineBytes+(i+2)];
    pixel[2]=lpSrc[(j-1)*1LineBytes+i];
    pixel[3]=lpSrc[(j-1)*1LineBytes+(i+1)];
    pixel[4]=lpSrc[(j-1)*1LineBytes+(i+2)];
    pixel[5]=lpSrc[j*1LineBytes+i];
    pixel[6]=lpSrc[j*1LineBytes+(i+1)];
    mean[5]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
        pixel[4]+pixel[5]+pixel[6])/7;
    var[5]=0;
    for(n=0;n<=6;n++)
        var[5]+=pixel[n]*pixel[n]-mean[5]*mean[5];

    //第7邻域
    pixel[0]=lpSrc[(j-2)*1LineBytes+(i-2)];
    pixel[1]=lpSrc[(j-2)*1LineBytes+(i-1)];
    pixel[2]=lpSrc[(j-1)*1LineBytes+(i-2)];
    pixel[3]=lpSrc[(j-1)*1LineBytes+(i-1)];
    pixel[4]=lpSrc[(j-1)*1LineBytes+i];
    pixel[5]=lpSrc[j*1LineBytes+(i-1)];
    pixel[6]=lpSrc[j*1LineBytes+i];
    mean[6]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
        pixel[4]+pixel[5]+pixel[6])/7;
    var[6]=0;
    for(n=0;n<=6;n++)
        var[6]+=pixel[n]*pixel[n]-mean[6]*mean[6];

    //第8邻域
    pixel[0]=lpSrc[j*1LineBytes+(i-1)];
    pixel[1]=lpSrc[j*1LineBytes+i];
    pixel[2]=lpSrc[(j+1)*1LineBytes+(i-2)];
    pixel[3]=lpSrc[(j+1)*1LineBytes+(i-1)];
    pixel[4]=lpSrc[(j+1)*1LineBytes+i];
    pixel[5]=lpSrc[(j+2)*1LineBytes+(i-2)];

```



```

        pixel[6]=lpSrc[(j+2)*lLineBytes+(i-1)];
mean[7]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
        pixel[4]+pixel[5]+pixel[6])/7;
        var[7]=0;
        for(n=0;n<=6;n++)
            var[7]+=pixel[n]*pixel[n]-mean[7]*mean[7];

        //第9邻域
        pixel[0]=lpSrc[j*lLineBytes+i];
        pixel[1]=lpSrc[j*lLineBytes+(i+1)];
        pixel[2]=lpSrc[(j+1)*lLineBytes+i];
        pixel[3]=lpSrc[(j+1)*lLineBytes+(i+1)];
        pixel[4]=lpSrc[(j+1)*lLineBytes+(i+2)];
        pixel[5]=lpSrc[(j+2)*lLineBytes+(i+1)];
        pixel[6]=lpSrc[(j+2)*lLineBytes+(i+2)];
mean[8]=(float)(pixel[0]+pixel[1]+pixel[2]+pixel[3]+
        pixel[4]+pixel[5]+pixel[6])/7;
        var[8]=0;
        for(n=0;n<=6;n++)
            var[8]+=pixel[n]*pixel[n]-mean[8]*mean[8];
        //求方差最小的邻域 nMin
        varMin=var[0];
        nMin=0;
        for(n=0;n<=8;n++)
            if(varMin>var[n])
            {
                varMin=var[n];
                nMin=n;
            }
        //将方差最小的邻域均值赋值给目标像素点
        lpDst[lLineBytes * j + i] = (BYTE)(mean[nMin]+0.5);
    }
}

```

#### (4) 中值滤波法

下面给出 3×3 中值滤波算法的核心代码：

```

int flag=1;                // 循环标志变量
BYTE pixel[10],mid;        // 窗口像素值及中值
BYTE temp;                 // 中间变量
// 中值滤波
for(j=1;j<lHeight-1;j++)
{
    for(i=1;i<lWidth-1;i++)
    {

```

```

//把 3×3 屏蔽窗口的所有像素值放入 pixel[m]
m=0;
for(y=j-1;y<=j+1;y++)
    for(x=i-1;x<=i+1;x++)
    {
        pixel[m]=lpSrc[y*lLineBytes+x];
        m++;
    }
//把 pixel[m] 中的值按降序排序
do{
    flag=0;
    for(m=0;m<9;m++)
        if(pixel[m]<pixel[m+1])
        {
            temp=pixel[m];
            pixel[m]=pixel[m+1];
            pixel[m+1]=temp;
            flag=1;
        }
}while(flag==1);
mid=pixel[4];
//将中值赋给目标图像的当前点
lpDst[lLineBytes * j + i] = mid;
}
}

```

## 5. 图像锐化模块

### (1) 梯度锐化

梯度锐化的核心代码如下：

```

BYTE t=20;
BYTE temp; //暂存双向一次微分结果

//逐个扫描图像中的像素点，进行门限梯度锐化处理
for(j=1;j<lHeight-1;j++)
{
    for(i=1;i<lWidth-1;i++)
    {
        //根据双向一次微分公式计算当前像素的灰度值
        temp=(BYTE)sqrt((float)((lpSrc[lLineBytes*j+i]-lpSrc[lLineBytes*j+(i-1)])*(lpSrc[lLineBytes*j+i]-lpSrc[lLineBytes*j+(i-1)])+
        (lpSrc[lLineBytes*j+i]-lpSrc[lLineBytes*(j-1)+i])*(lpSrc[lLineBytes*j+i]-lpSrc[lLineBytes*(j-1)+i])));
        if (temp>=t)
        {
            if((temp+100)>255)

```

```

        lpDst[lLineBytes*j+i]=255;
    else
        lpDst[lLineBytes*j+i]=temp+100;
    }
    if (temp<t)
        lpDst[lLineBytes*j+i]=lpSrc[lLineBytes*j+i];
    }
}

```

## (2) 拉普拉斯掩膜锐化

拉普拉斯算子可以用图 7-9a 的模板形式表示,图 7-9b 表示拉普拉斯算子的扩展模板。

其核心代码如下:

0 1 0	1 1 1
1 -4 1	1 -8 1
0 1 0	1 1 1

a) 拉普拉斯算子      b) 扩展模板

图 7-9 拉普拉斯算子模板及其扩展模板

```

int tempH;           // 模板高度
int tempW;           // 模板宽度
float tempC;         // 模板系数
int tempMY;          // 模板中心元素 Y 坐标
int tempMX;          // 模板中心元素 X 坐标
float Template[9];   // 模板数组
// 设置拉普拉斯模板参数
tempW=3;
tempH=3;
tempC=1.0;
tempMY=1;
tempMX=1;
Template[0]=-1.0;
Template[1]=-1.0;
Template[2]=-1.0;
Template[3]=-1.0;
Template[4]=9.0;
Template[5]=-1.0;
Template[6]=-1.0;
Template[7]=-1.0;
Template[8]=-1.0;
// 调用卷积函数
Convolution(tempH,tempW,tempMX,tempMY,Template,tempC);

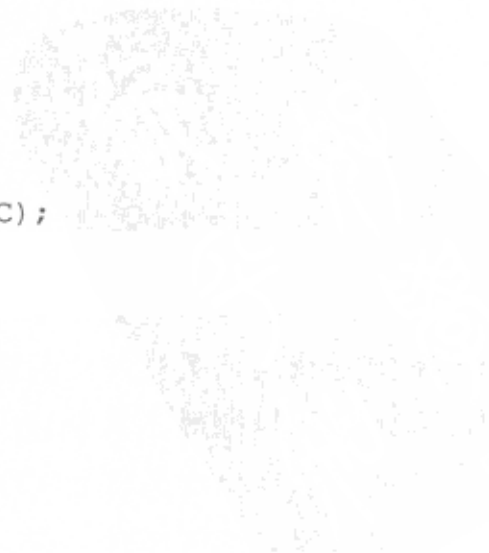
```

## Convolution 中的核心代码:

```

// 逐个扫描图像中的像素点,进行卷积运算
for(j=tempMY;j<lHeight-tempH+tempMY+1;j++)
{
    for(i=tempMX;i<lWidth-tempW+tempMX+1;i++)
    {
        // 计算像素值
    }
}

```



```

fResult=0;
for(k=0;k<tempH;k++)
    for(l=0;l<tempW;l++)
        fResult=fResult+lpSrc[(j-tempMY+k)*lLineBytes+(i-tempMX+l)]
            *fpTempArray[k*tempW+l];
// 乘上系数
fResult*=fCoef;
// 取绝对值
fResult=(float) fabs(fResult);
// 判断是否超过 255
if(fResult>255)
    // 若超过 255, 直接赋值为 255
    lpDst[j*lLineBytes+i]=255;
else
    // 未超过 255, 赋值为计算结果
    lpDst[j*lLineBytes+i]=(BYTE) (fResult+0.5);
}
}

```

## 6. 伪彩色增强模块

伪彩色增强变换是用指定伪彩色调色板来替换当前图像调色板实现的, 其代码如下:

```

DWORD wNumColors; // 颜色表中的颜色数目
wNumColors =m_pDibResult->m_lpBMiH->biBitCount; // 获取原图像颜色表中颜色数目
LPRGBQUAD m_lpRgbQuad;
m_lpRgbQuad=(LPRGBQUAD) (m_pDibResult->m_lpvColorTable);

if (wNumColors == 8) // 判断颜色数目是否为 256 色
    // 读取伪彩色编码, 更新调色板
    for (i = 0; i < 256; i++)
    {
        // 更新调色板红色、绿色、蓝色分量
        (m_lpRgbQuad+i)->rgbBlue = ColorsTable[8][i][0];
        (m_lpRgbQuad+i)->rgbGreen = ColorsTable[8][i][1];
        (m_lpRgbQuad+i)->rgbRed = ColorsTable[8][i][2];
        // 更新调色板保留位
        (m_lpRgbQuad+i)->rgbReserved = 0;
    }
}

```

### 7.4.4 遥感图像配准模块

在主窗口中打开遥感图像后, 用户可以通过单击【图像配准】菜单打开【图像配准】对话框。由于图像配准与图像几何校正原理基本一致, 唯一不同之处在于图像配准有半自动特征点选取模式, 因此, 其他原理不再赘述, 本节只对半自动特征点选取模式进行分析。

特征点的半自动选取用 CDlgReg::auto\_Featurechosen()函数实现。

同手动模式选取函数一样, 本系统需要进行以下三步来判断选中的点是否可用。

- 1) 计算特征点合法区域 (即待配准图像上比图像区域小一圈的区域), 判断选取的点是否合法。
- 2) 判断选取的点是否为以前的特征点, 如果是则将其去掉, 并将对应的基准图像 (或待校正图像) 上的点也去掉。
- 3) 判断特征点是否已选够, 如果特征点选取足够则弹出提示消息 “已经选取了 3 个特征点, 如果要继续选取, 可以去掉配准不正确的特征点再进行选取”。

如果经过以上判断此点可用, 则将此点进行保存并对此点进行匹配, 即在基准图像中找出此点的同名点。CDlgReg::auto\_Featurechosen()函数代码如下:

```
void CDlgReg::auto_Featurechosen(UINT nFlags, CPoint point)
{
    int i, j;                // 循环变量
    // 如果特征选取标志位为 TRUE, 则进行特征点的选取和配准, 否则退出
    if(!m_bChoseFeature)
        return;
    //待配准图像的特征选取区域, 在这里选择特征点的区域要比图像的区域小一圈
    CRect rectChoose;
    rectChoose.bottom = m_rectResltImage.bottom - 5;
    rectChoose.top = m_rectResltImage.top + 5;
    rectChoose.left = m_rectResltImage.left + 5;
    rectChoose.right = m_rectResltImage.right - 5;

    CRect rectFeature;        // 特征点的区域
    BOOL bFlag = FALSE;      // 标志位, 表示此点是否为已经选择的特征点

    // 判断此点是否合法, 并判断此点是否已经选择, 如果是, 则去掉此点
    if(rectChoose.PtInRect(point))
    {
        // 如果所选择的特征点是以前的特征点, 则去掉此点
        for( i = 0; i<m_nChsFeatureNum; i++)
        {
            // 选择特征点的显示区域, 以便对特征点进行取舍
            rectFeature.bottom = m_pPointSampl[i].y + m_rectResltImage.top + 5;
            rectFeature.top = m_pPointSampl[i].y + m_rectResltImage.top - 5;
            rectFeature.left = m_pPointSampl[i].x + m_rectResltImage.left - 5;
            rectFeature.right = m_pPointSampl[i].x + m_rectResltImage.left + 5;

            // 判断所选择的特征点是否为原来选择的特征点
            // 如果是, 则去掉此特征点
            if(rectFeature.PtInRect(point))
            {
                // 将后面的特征点向前移动一位, 去掉所选择的特征点
                for(j=i; j<m_nChsFeatureNum-1; j++)
                {
                    m_pPointSampl[j] = m_pPointSampl[j+1];
                    m_pPointBase[j] = m_pPointBase[j+1];
                }
            }
        }
    }
}
```

```

    }
    m_nChsFeatureNum--; // 将特征点的计数减一
    Invalidate();       // 更新显示
    bFlag = TRUE;       // 设置标志位
    return;
}

// 再判断特征点是否已经选够了
if(m_nChsFeatureNum == 3)
{
    AfxMessageBox("已经选取了3个特征点, 如果要继续选取,
        可以去掉配准不正确的特征点再进行选取");
    return;
}

// 如果此点是需要选取的, 则进行相关操作
if(!bFlag)
{
    // 将此待配准特征点选取, 注意特征点的坐标是以图像的左上角为原点确定的
    m_pPointSampl[m_nChsFeatureNum].x = point.x - m_rectResltImage.left;
    m_pPointSampl[m_nChsFeatureNum].y = point.y - m_rectResltImage.top;
    // 配准此特征点
    m_pPointBase[m_nChsFeatureNum] = FindMatchPoint(m_pDibInit, m_pDibSamp,
        m_pPointSampl[m_nChsFeatureNum]);
    m_nChsFeatureNum++; // 将特征点计数加一
}
}
}

```

由上面的代码可以看出来, auto\_Featurechosen 是调用 FindMatchPoint 进行特征点配准的。该函数根据待配准图像中的特征点位置在基准图像中寻找匹配特征点, 并将匹配的特征点位置返回。在配准的过程中, 采取的是块配准的方法进行匹配, 在这里选取块的大小为  $7 \times 7$ 。搜索的方法为全局搜索。具体代码如下:

```

CPoint CDlgReg::FindMatchPoint(CDib* pDibBase, CDib* pDibSamp, CPoint pointSamp)
{
    int i, j, m, n;                // 循环变量
    int nX, nY;                    // 临时变量
    CPoint pointBase;              // 配准特征点位置
    int nBlockLen = 7;             // 配准数据块的尺寸
    int nBlockHalfLen = 3;

    unsigned char* pBase;          // 基准图像数据指针
    pBase = (unsigned char *)pDibBase->m_lpImage;
    unsigned char* pSamp;          // 待配准图像数据指针
    pSamp = (unsigned char *)pDibSamp->m_lpImage;
}

```



```

unsigned char* pUnchSampBlock;    // 特征点位置的数据配准块
pUnchSampBlock = new unsigned char[nBlockLen*nBlockLen];
unsigned char* pUnchBaseBlock;    // 临时分配内存, 用于存放配准数据块
pUnchBaseBlock = new unsigned char[nBlockLen*nBlockLen];

double dbCor;                    // 相似度
double dbMaxCor = 0;             // 最大相似度
CSize sizeBaseImg;               // 基准图像的存储大小
sizeBaseImg = pDibBase->GetDibSaveDim();
CSize sizeSampImg;               // 待配准图像的存储大小
sizeSampImg = pDibSamp->GetDibSaveDim();

// 从待配准图像中提取以特征点为中心的 nBlockLen*nBlockLen0 的数据块
for(i=-nBlockHalfLen; i<=nBlockHalfLen; i++){
    for(j=-nBlockHalfLen; j<=nBlockHalfLen; j++){
        // 计算此点在图像中的位置
        nX = pointSamp.x + i;
        nY = sizeSampImg.cy - pointSamp.y + j + 1;
        // 提取图像数据
        pUnchSampBlock[(j+nBlockHalfLen)*nBlockLen + (i+nBlockHalfLen)] =
            pSamp[nY*sizeSampImg.cx + nX];
    }
}

// 基准图像的高度和宽度
int nBaseImgHeight, nBaseImgWidth;
nBaseImgHeight = pDibBase->m_lpBMIH->biHeight;
nBaseImgWidth = pDibBase->m_lpBMIH->biWidth;

// 在基准图像中寻找配准特征点, 采取的搜索方法为全局搜索
for(m = nBlockHalfLen; m< nBaseImgHeight-nBlockHalfLen; m++){
    for(n=nBlockHalfLen; n<nBaseImgWidth-nBlockHalfLen; n++){
        // 提取以此点为中心, 大小为 nBlockLen*nBlockLen 的数据块
        for(i=-nBlockHalfLen; i<=nBlockHalfLen; i++){
            for(j=-nBlockHalfLen; j<=nBlockHalfLen; j++){
                // 计算此点在图像中存储的位置
                nX = n + i;
                nY = sizeBaseImg.cy - m + j + 1;
                // 提取图像数据
                pUnchBaseBlock[(j+nBlockHalfLen)*nBlockLen + (i+nBlockHalfLen)] =
                    pBase[nY*sizeBaseImg.cx + nX];
            }
        }
        // 对这两个数据块进行配准, 计算相似度
        dbCor = CalCorrelation(pUnchBaseBlock, pUnchSampBlock, nBlockLen);
        // 判断是否为最大相似度, 如果是, 则记录此相似度和配准特征点位置
        if(dbCor > dbMaxCor){

```

```

        dbMaxCor = dbCor;
        pointBase.x = n;
        pointBase.y = m;
    }
}
return pointBase;
}

```

可以看出,上面代码调用 CDlgReg::CalCorrelation 计算两个数据块的相似度,其中,去掉均值以消除亮度变换的影响。代码如下:

```

double CDlgReg::CalCorrelation(unsigned char* pBase, unsigned char* pSamp, int
nBlockLen)
{
    double dbSelfBase=0,dbSelfSamp=0;    // 临时变量
    double dbCor=0;                      // 相似度
    double dbMeanBase=0,dbMeanSamp=0;    // 块均值
    // 计算两个块的平均值
    for(int i=0;i<nBlockLen;i++)
        for(int j=0;j<nBlockLen;j++)
        {
            dbMeanBase += pBase[j*nBlockLen + i];
            dbMeanSamp += pSamp[j*nBlockLen + i];
        }
    dbMeanBase = dbMeanBase/(nBlockLen*nBlockLen);
    dbMeanSamp = dbMeanSamp/(nBlockLen*nBlockLen);
    // 求取相似度
    for(int i=0;i<nBlockLen;i++)
        for(int j=0;j<nBlockLen;j++)
        {
            dbSelfBase += (pBase[j*nBlockLen + i] - dbMeanBase)*
                (pBase[j*nBlockLen + i] - dbMeanBase);
            dbSelfSamp += (pSamp[j*nBlockLen + i] - dbMeanSamp)*
                (pSamp[j*nBlockLen + i] - dbMeanSamp);
            dbCor += (pBase[j*nBlockLen + i] - dbMeanBase)*
                (pSamp[j*nBlockLen + i] - dbMeanSamp);
        }
    dbCor = dbCor / sqrt(dbSelfBase * dbSelfSamp);
    return dbCor;    // 返回相似度
}

```

## 7.5 经验分享

下面是在本系统开发过程中遇到的和考虑到的一些问题及解决方案,可能会有助于读者的学习和编程实践。

1) 本系统是基于 Windows 7 操作系统和 VC 2008 平台开发的, 并重用了一些 VC 6.0 的代码, 由于 VC 2008 与 VC 6.0 平台之间的差异, 系统编译时遇到了问题。语句“throw new CException”在编译的时候显示错误为——“error C2259: “CException”: 不能实例化抽象类。”

这是因为在 VC 2008 中 CException 是一个抽象类, 它无法实例化。解决方案是加入一个命名为 CMyException.h 的头文件:

```
#pragma once    // 保证头文件只被编译一次
class CMyException :public CException
{
};
```

然后把程序中的其他 CException 全部替换为 CMyException。

2) 本系统在编译过程中有时会出现如下错误:

“LINK : fatal error LNK1000: Internal error during IncrBuildImage.”

这是 VC 的一个内部 BUG, 可以下载 KB948127 补丁来解决, 如果没有下载补丁, 则可以通过以下方法解决: 选择【项目】/【属性】命令, 打开【遥感图像配准系统 属性页】对话框, 选择左栏的【配置属性】/【链接器】命令, 可以看到右栏第四行是“启用增量链接”, 将此项中的“是(/INCREMENTAL)”改为“否(/INCREMENTAL:NO)”。

3) 本系统在几何校正模块和遥感图像配准模块中均使用了二元一次多项式, 由于 GCP 最小选取数目为  $(n+1)(n+2)/2$ , 因此本系统选了 3 个 GCP。系统本身已经具有较高的精确度, 但是读者可以通过增加 GCP 的选取个数提高系统图像处理结果的精确度。

4) 从提高几何校正结果精确度的角度看, 在几何校正模块中, 基准图像最好采用与待校正图像同比例尺的地图。但由于此方面数据的不足, 本系统使用了自制的基准图像(利用边缘提取技术提取的相应图像), 在条件允许的情况下, 读者可以使用标准图像作为基准图像进行几何校正, 从而取得更好的校正效果。

5) 从提高程序适用性的角度看, 本系统在遥感图像配准模块中设置了两种 GCP 的选取模式: 半自动模式和手动模式。手动模式在任何情况下均可以使用, 而半自动模式在图像效果较差的情况下不宜使用。另一方面, 由于手动模式是在用户肉眼观察下进行的, 因此其精确度低于半自动模式。故而在图像效果较好的情况下, 用户可以优先选择半自动模式, 在半自动模式下选取特征点, 少数情况下其对应的同名点可能匹配错误(这种错误很容易用肉眼识别), 这时用户可以自己取消半自动匹配错误的 GCP。当这种错误过多时, 即图像效果较差时, 用户可以由半自动模式切换为手动模式。

6) 图像进行几何校正需要使用二元  $n$  次多项式。本系统取  $n=1$ , 即采用二元一次多项式进行几何校正, 可以进行线性的坐标变换, 解决比例尺、中心移动、歪斜等方面的几何畸变。因此, 要想提高程序的适用性, 读者可以取  $n \geq 2$ , 以便解决传感器偏航、俯仰、滚动等因素引起的更复杂的几何畸变。

## 第 8 章 图像检索系统

“众里寻他千百度，蓦然回首，那人却在灯火阑珊处。”是词人辛弃疾笔痕墨影间的一种人生境界，如果用计算机视觉这个行当里的术语来刻画这种千寻百觅后不经意的发现，那就是图像检索。

自 20 世纪 70 年代以来，在数据库系统和计算机视觉两大研究领域的共同推动下，图像检索技术已逐渐成为一个非常活跃的研究领域，已从最初基于文本的图像检索发展到基于内容的图像检索。基于内容的图像检索融合了模式识别、计算机视觉以及图像理解等技术，利用图像的视觉特征（如图像的颜色、纹理以及形状等）进行检索。本章介绍图像检索中常用的特征提取与相似度计算技术，以及基于内容的图像检索系统的实现方法。

**本章要点：**

- 图像特征提取技术
- 相似度计算技术
- 图像检索系统功能描述
- 图像检索系统的总体结构和主要流程
- 图像检索系统的编程实现

### 8.1 核心技术原理

图像检索系统涉及的核心技术主要有图像特征提取技术和相似度计算技术。前者解决的是以什么为依据来判定图像的相似性，后者解决的是图像之间到底有多相似的问题。


#### 8.1.1 图像特征提取技术

图像特征的表达与提取是基于内容的图像检索技术的基础。通常，基于内容的图像检索主要是依据图像的视觉特征，如图像的颜色、纹理、形状以及空间关系等特征进行检索的。因此图像特征的提取主要包括对颜色特征、纹理特征、形状特征以及空间关系特征的提取。而对某个特定的图像特征，通常又有多种不同的表达方法。由于人们主观认识上的千差万别，对于某个特征并不存在一个所谓的最佳的表达方式。然而图像特征的不同表达方式从多个不同的角度刻画了该特

征的某些性质。

### 1. 颜色特征

颜色特征是人类区分与感知不同物体的最基本的视觉特征,是图像最底层、最直观的一种全局的物理特征,描述了图像或图像区域所对应的景物的表面性质。在许多情况下,颜色是描述一幅图像最简单、最有效的特征。同时,颜色特征也是图像检索中使用广泛且比较可靠的一种视觉特性,因为与其他的视觉特征相比,颜色特征具有一定的稳定性,通常对旋转、平移、噪声、图像质量的退化、尺寸、分辨率和方向都不敏感,表现出很强的鲁棒性。Swain 等提出的基于颜色直方图的方法是最早的采用颜色特征进行图像检索的方法。

 对颜色特征的提取都是在某个颜色空间里进行的。常用的颜色空间有 RGB、CIELAB 以及 HSV 等。采用哪种颜色空间并没有固定标准,一般都是根据处理图像时所关注的颜色特征能否从颜色空间的某一分量直接获得来确定。本章采用 HSV 颜色空间。

常用的颜色特征提取方法主要包括颜色直方图、累计直方图、颜色矩、颜色集和颜色聚合向量等方法。

#### (1) 颜色直方图

颜色直方图 (Color Histogram) 是基于颜色特征的图像检索技术中最常用的特征表示方法,其优点是不受图像旋转和平移变化的影响,进一步借助归一化,还可不受图像尺度变化的影响,其缺点是没有表达出颜色空间分布的信息。

对一幅数字图像,统计每一种颜色在该图像中出现的像素点数,以颜色值为横坐标,以颜色出现的像素点数为纵坐标,据此绘出的图形即是该图像的颜色直方图。图像颜色直方图  $H$  定义如式 (8-1) 所示:

$$H = \left\{ (h[c_1], h[c_2], \dots, h[c_k]) \mid \sum_{k=1}^n h[c_k] = 1, 0 \leq h[c_k] \leq 1 \right\} \quad (8-1)$$

第  $k$  种颜色在图像中出现的像素点频数用  $h[c_k]$  表示,计算公式 (8-2) 如下:

$$h[c_k] = \frac{\sum_{i=0}^{N_1-1} \sum_{j=0}^{N_2-1} \begin{cases} 1 & (\text{当 } I(i, j) = c_k) \\ 0 & (\text{其他}) \end{cases}}{N_1 \times N_2} \quad (8-2)$$

其中  $N_1$  和  $N_2$  表示图像中的宽和高。

#### (2) 累计直方图

如果图像特征向量不能取到所有可能值,许多零值会出现在颜色直方图中,从而影响直方图相交运算,导致匹配结果不能合理地反映图像间的颜色差别。有人提出用累计直方图来解决这个问题。

累计直方图的定义如式 (8-3):

$$H(K) = \frac{\sum_{i=1}^k n_i}{N}, k = 0, 1, \dots, L-1 \quad (8-3)$$

### (3) 颜色矩

由 Stricker 和 Orengo 所提出的颜色矩 (Color Moments) 也是一种简单、有效的颜色特征。这种方法的数学基础在于图像中任何的颜色分布均可以用它的矩来表示。此外, 由于颜色分布信息主要集中在低阶矩中, 因此仅采用颜色的一阶矩、二阶矩和三阶矩就足以表示图像的颜色分布。与颜色直方图不同的是, 颜色矩不需要对特征进行向量化。颜色一、二、三阶矩如式 (8-4)、式 (8-5) 以及式 (8-6) 所示:

$$\mu_i = \frac{1}{N} \sum_{j=1}^N p_{ij} \quad (8-4)$$

$$\delta_i = \left[ \frac{1}{N} \sum_{j=1}^N (p_{ij} - \mu_i)^2 \right]^{\frac{1}{2}} \quad (8-5)$$

$$s_i = \left[ \frac{1}{N} \sum_{j=1}^N (p_{ij} - \mu_i)^3 \right]^{\frac{1}{3}} \quad (8-6)$$

上式中, 像素数目为  $N$ ,  $p_{ij}$  为第  $j$  个像素的第  $i$  个颜色分量。与其他的颜色特征相比, 图像的颜色矩只有 9 个分量 (3 个颜色分量, 每个分量有 3 个低阶矩)。低阶矩一般起到过滤缩小范围的作用, 因为它的分辨能力不够, 常和其他特征配合使用。

### (4) 颜色集


为支持大规模图像库中的快速查找, Smith 和 Chang 提出了用颜色集 (Color Sets) 作为对颜色直方图的一种近似。他们将颜色空间量化成若干个 bin。然后, 用色彩自动分割技术将图像分为若干区域, 每个区域用量化颜色空间的某个颜色分量来索引, 从而将图像表达为一个二进制的颜色索引集。在图像匹配中, 比较不同图像颜色集之间的距离和色彩区域的空间关系 (包括区域的分离、包含、交等, 每种空间关系对应不同的评分)。因为颜色集表达为二进制的特征向量, 可以构造二分查找树来加快检索速度, 这对于大规模的图像集合十分有利。

### (5) 颜色聚合向量

针对颜色直方图和颜色矩无法表达图像色彩的空间位置的缺点, Pass 提出了图像的颜色聚合向量 (Color Coherence Vector)。它是颜色直方图的一种演变, 其核心思想是将属于直方图每一个 bin 的像素分为两部分。如果该 bin 内的某些像素所占据的连续区域的面积大于给定的阈值, 则将该区域内的像素作为聚合像素, 否则作为非聚合像素。由于包含了颜色分布的空间信息, 颜色聚



合向量相比颜色直方图可以达到更好的检索效果。

 本章中的图像检索系统使用颜色直方图、累计直方图以及颜色矩这3种方法分别进行颜色特征的提取。

## 2. 纹理特征

纹理特征是所有物体表面所固有的特性,也是一种全局特征,包含了物体表面的组织结构排列的重要信息。在自然界中,如砖、纺织物等都有各自独特的纹理特征。在计算机视觉研究中,与颜色特征不同,纹理特征不是基于像素点的特征,它需要在包含多个像素点的区域中进行统计计算,因此纹理特征具有一种不依赖于颜色或亮度的反映图像中同质现象的视觉特征,可以从微观上区分图像中不同的物体。在模式匹配中,这种区域性的特征具有较大的优越性,不会由于局部的偏差而无法匹配成功。作为一种统计特征,纹理特征常具有旋转不变性,并且对于噪声有较强的抵抗能力。因此,纹理特征是基于内容的图像检索系统中常用的特征。

### (1) 灰度共生矩阵

灰度共生矩阵是 Haralick 于 1973 年提出的纹理特征提取方法,它是应用最早的一种纹理特征提取方法。灰度共生矩阵主要是估计图像的二阶组合条件概率密度函数。

灰度共生矩阵定义为在  $\theta$  方向上,相隔  $d$  像元距离的一对像元,具有灰度值  $i$  和  $j$  的概率,记为  $p(i, j | d, \theta)$ 。如果  $\theta$  和  $d$  确定,可简记为  $p_{i,j}$ 。灰度共生矩阵是对称矩阵,它的阶数由图像的灰度级数决定。灰度共生矩阵每个元素的值由式 (8-7) 确定:

$$p(i, j | d, \theta) = \frac{p(i, j | d, \theta)}{\sum_i \sum_j p(i, j | d, \theta)} \quad (8-7)$$

灰度共生矩阵主要考虑像素灰度值的相关性,并用一对像素出现某种灰度值的条件概率来表示纹理,灰度共生矩阵可以定量地描述纹理特征。

灰度共生矩阵中,总共有 14 种特征来表征图像的纹理,但是在实际应用中计算 14 种特征,计算量显然很大。所以一般情况下,只选用能量、熵、惯性矩以及相关这 4 个特征。

#### 1) 能量:

$$ASM = \sum_i \sum_j p(i, j)^2 \quad (8-8)$$

主要用来度量灰度值分布的均匀性。

#### 2) 熵:

$$ENT = -\sum_i \sum_j p(i, j) \log(p(i, j)) \quad (8-9)$$

如果熵值较大则表示共生矩阵的值较分散;如果熵值较小,则表示共生矩阵的值较集中。

3) 惯性矩:

$$CON = \sum_i \sum_j (i-j)^2 p(i, j) \quad (8-10)$$

如果惯性矩值较小则表示共生矩阵较集中于主对角线附近, 如果惯性矩值较大则表示纹理为细纹理。

4) 相关:

$$COR = \frac{\sum_i \sum_j ijp(i, j) - \mu_x \mu_y}{\sigma_x \sigma_y} \quad (8-11)$$

其中,  $\mu_x$ 、 $\mu_y$ 、 $\sigma_x$  以及  $\sigma_y$  分别表示  $p_x$  和  $p_y$  的均值和标准差。共生矩阵中行列元素的相似度用相关来描述, 以此来度量灰度的线性关系。

(2) Tamura 纹理特征

基于人类对纹理的视觉感知心理学研究, Tamura 等提出了另一种纹理特征的表示方法: Tamura 纹理特征。它一共有 6 个分量, 对应于纹理特征 6 个属性, 分别为对比度 (Contrast)、方向度 (Directionality)、粗糙度 (Coarseness)、线像度 (Line Likeness)、规整度 (Regularity) 和粗略度 (Roughness)。其中前 3 个属性对于图像检索尤其重要。

1) 粗糙度可以通过以下几步计算得到:

首先, 利用式 (8-12) 计算图像中大小为  $2^k \times 2^k$  个像素的活动窗口中像素的平均灰度值。

$$A_k(x, y) = \sum_{i=x-2^{k-1}}^{x+2^{k-1}-1} \sum_{j=y-2^{k-1}}^{y+2^{k-1}-1} g(i, j) / 2^{2k} \quad (8-12)$$

其中  $k = 0, 1, \dots, 5$ ,  $g(i, j)$  是位于  $(i, j)$  的像素灰度值。

其次, 对于每个像素, 分别计算它在水平和垂直方向上互不重叠的窗口之间的平均灰度差。

$$\begin{aligned} E_{k,h}(x, y) &= |A_k(x+2^{k-1}, y) - A_k(x-2^{k-1}, y)| \\ E_{k,v}(x, y) &= |A_k(x, y+2^{k-1}) - A_k(x, y-2^{k-1})| \end{aligned} \quad (8-13)$$

其中对于每个像素, 能使  $E$  值达到最大的  $k$  值用来设置最佳尺寸  $S_{best}(x, y) = 2^k$ 。

最后, 粗糙度可以通过计算整幅图像中  $S_{best}$  的平均值来得到, 表达式为:

$$F_{crs} = \frac{1}{m \times n} \sum_{i=1}^m \sum_{j=1}^n S_{best}(i, j) \quad (8-14)$$

2) 对比度是通过对像素灰度分布情况的统计得到的:

$$F_{con} = \frac{\sigma}{\alpha_4^{1/4}} \quad (8-15)$$

其中  $\alpha_4 = \frac{\mu_4}{\sigma^{1/4}}$ ,  $\mu_4$  是四次矩,  $\sigma^2$  是方差。

3) 方向度则需要通过以下 3 步进行计算:

首先, 计算每个像素处的梯度向量。

其次, 构造直方图  $H_D$  用来表达  $\theta$  值, 该直方图首先对  $\theta$  的值域范围进行离散化, 然后统计每个 bin 中相应的梯度向量的模大于给定阈值的像素数量。这个直方图对于具有明显方向性的图像会表现出峰值, 对于无明显方向性的图像则表现得比较平坦。

最后, 图像总体的方向性可以通过计算直方图中峰值的尖锐程度获得, 表达式如下:

$$F_{\text{dir}} = \sum_p \sum_{\phi \in w_p} (\phi - \phi_p)^2 H_D(\phi) \quad (8-16)$$

其中  $p$  代表直方图中的峰值,  $n_p$  代表直方图中所有的峰值。对于某个峰值  $p$ ,  $w_p$  代表该峰值所包含的所有 bin, 而  $\phi_p$  是具有最高值的 bin。

### (3) 自回归纹理模型

应用随机场模型表示纹理特征是近年来许多学者的研究重点, 其中马尔可夫随机场模型获得了巨大的成功。马尔可夫随机场模型的一种应用实例就是自回归纹理模型 (SAR)。在 SAR 模型中, 每个像素的灰度用随机变量表示, 并可以通过其相邻像素来描述。

如用  $s$  代表某个像素, 其灰度值  $g(s)$  可以用其相邻的像素灰度值的线性叠加和噪音项  $\varepsilon(s)$  的和来表示:

$$g(s) = \mu + \sum_{r \in D} \theta(r) g(s+r) + \varepsilon(s) \quad (8-17)$$

其中  $\mu$  是基准偏差, 由整幅图像的平均灰度值决定,  $D$  表示  $s$  的相邻像素集。  $\theta(r)$  是一系列模型参数, 表示不同相邻位置上的像素的权值。  $\varepsilon(s)$  是均值为 0 而方差为  $\sigma^2$  的高斯随机变量。通过上式可以用回归法计算参数  $\theta$  和标准方差  $\sigma$  的值, 它们反映了图像的各种纹理特征。

### (4) 基于小波变换的纹理特征

纹理分析方法的另一种重要方法是小波变换。小波变换将信号分解为一系列的基本函数  $\psi_{mn}(x)$ , 这些基本函数可以通过母函数  $\psi(x)$  变换得到, 如式 (8-18):

$$\psi_{mn}(x) = 2^{-m/2} \psi(2^{-m}x - m) \quad (8-18)$$

式中,  $m$  和  $n$  是整数。因此, 信号  $f(x)$  可以表示为:

$$f(x) = \sum_{m,n} c_{mn} \psi_{mn}(x) \quad (8-19)$$

在计算二维小波变换时, 在每个层次上, 二维信号被分解为 4 个子波段, 根据其频率特性命名为 LL、LH、HL 和 HH。可以用于纹理分析的主要有两种类型的小波变换, 分别是金字塔结构的小波变换 (Pyramid-structured Wavelet Transform) 和树状结构的小波变换 (Tree-structured Wavelet Transform)。小波变换表示的纹理特征可以用每个波段的每个分解层次上能量分布的标准差和均值来表示。

PWT 递归分解 LL 波段。但是, 如果纹理特征的主要信息在中频段范围内, 就不能简单地只分解低频的 LL 波段。不过, TWT 可用来解决此问题。因为 TWT 除了递归分解 LL 波段外, 还分解其他的 LH、HL 和 HH 波段。

### 3. 形状特征

形状特征是刻画物体的本质特征之一, 更能符合人们的视觉感知特性, 并具有不受目标颜色、纹理以及背景变化影响等特点。形状特征的重要原则是对位移、旋转和尺度变换的不变性, 因为人类出于识别和检索的目的, 总是趋向于忽略这些变化。但不同于颜色和纹理等低层特征, 形状特征的表达必须以对图像中物体或区域的划分为基础, 以图像分割为前提。因此, 相对于颜色和纹理特征来说, 它的提取和描述比较困难。

通常对于形状的描述可以分为两种, 基于边界的 (Boundary-based) 和基于区域的 (Region-based), 前者只利用形状的外部边缘, 而后者利用形状的全部区域。

基于边界的形状特征提取关键在于边缘检测的研究, 在提取边缘的基础上, 定义边缘的特征描述, 常见的有周长、链码、边界分段序列、曲率、样条拟合曲线、兴趣点、傅里叶形状描述子、数学形态描述以及神经网络识别等。其中傅里叶形状描述子是最典型的方法。

基于区域的表示方法主要是通过图像分割技术提取出图像中用户感兴趣的物体, 依靠区域内像素的分布信息提取图像特征。因为其将区域形状当做一个整体来看待, 有效地利用了区域内的所有像素信息, 因而受噪声和形状变化的影响相对较小。形状的区域特征主要有区域的面积、离散度、欧拉数、偏心率、区域骨架、区域不变矩、Legendre 矩、几何不变矩、Zernike 矩、伪 Zernike 矩、旋转矩、通用傅里叶描述子、复数矩以及角半径变换等方法。其中不变矩是最常用的一种方法。

#### (1) 傅里叶形状描述子

傅里叶形状描述子 (Fourier Shape Descriptors) 的最主要优点是起始点无关, 转置不变性和旋转不变性等。傅里叶形状描述子的基本思想是假定物体的形状是一条封闭的曲线, 沿边界曲线上的一个动点的坐标变化是一个以形状边界周长为周期的函数。这个周期函数可以展开为傅里叶级数, 而傅里叶级数中一系列系数直接与边界曲线的形状有关, 可用于描绘形状, 因此称为傅里叶形状描述子。

令  $L$  表示区域  $P$  的边界,  $s$  表示从  $L$  上的起始点  $b_0$  到沿曲线  $L$  逆时针方向上某一动点  $b$  之间的弧长。  $S$  表示曲线  $L$  的周长。动点  $b$  的坐标  $(x(s), y(s))$  既是  $x, y$  的函数又是弧长的函数, 因此曲线的参数方程可用复数形式表示为:

$$L(s) = x(s) + iy(s) \quad (8-20)$$

它是一个周期函数,即

$$L(s+S)=L(s), 0 \leq s \leq S \quad (8-21)$$

令  $t=2\pi s/S$ , 则上式可以写成:

$$L(t)=x(t)+iy(t) \quad (8-22)$$

$L(t)$  就是以  $2\pi$  为周期的周期函数, 它的傅里叶展开式为:

$$L(t)=\sum_{n=-\infty}^{+\infty} C_n e^{int} \quad (8-23)$$

其中  $C_n$  为傅里叶系数, 有

$$C_n=\frac{1}{2\pi} \int_0^{2\pi} L(t) e^{-int} dt, n=0, \pm 1, \pm 2 \dots \quad (8-24)$$

物平面内的平移只影响边界傅里叶系数的直流分量, 而旋转和边界的初始点选取只影响傅里叶系数的相位, 因此取前  $m$  个傅里叶变换低频系数的幅度, 去掉直流分量, 并对  $L(t)$  进行归一化, 可以作为边界的特征描述, 该特征对平移、旋转、尺度变换和初始点选取是不敏感的。

由此可以导出如下形状描述子:

$$f_F\left(\frac{|L(-m/2)|}{|L(1)|}, \dots, \frac{|L(-1)|}{|L(1)|}, \frac{|L(2)|}{|L(1)|}, \dots, \frac{|L(m/2)|}{|L(1)|}\right) \quad (8-25)$$

## (2) 不变矩

1962 年, Hu 首次提出了基于代数不变量的矩不变量, 并通过对几何矩的非线性组合, 导出了一组对于图像平移、旋转和尺度变化不变的矩。不变矩是图像的一种统计特征, 它利用图像灰度分布的各阶矩来描述图像灰度的分布特性。

对于离散的数字图像  $f(x, y)$  的  $p+q$  阶中心矩定义为:

$$\mu_{pq}=\sum_x \sum_y (x-\bar{x})^p (y-\bar{y})^q f(x, y) \quad (8-26)$$


其中,  $(\bar{x}, \bar{y})$  表示图像的区域中心。中心矩表示了图像内不同灰度级的像素相对于其中心是如何分布的, 因此中心矩具有位置无关性。为了获取针对图像缩放无关的性质, 可以对该中心矩进行如下标准化操作:

$$\eta_{pq}=\frac{\mu_{pq}}{\mu_{00}^\gamma} \quad (8-27)$$

其中  $\gamma=\frac{p+q}{2}+1, p+q=2, 3, \dots$ 。

基于标准化的二阶和三阶中心矩, 可以导出 HU 不变矩如下:

$$\left. \begin{aligned}
 \phi_1 &= \eta_{20} + \eta_{02} \\
 \phi_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
 \phi_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
 \phi_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
 \phi_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\
 &\quad + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
 \phi_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
 &\quad + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
 \phi_7 &= (3\eta_{21} - \eta_{03})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\
 &\quad + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
 \end{aligned} \right\} \quad (8-28)$$

 本章中的图像检索系统使用不变矩的方法进行形状特征的提取。

#### 4. 空间关系特征

上述的颜色、纹理和形状等多种特征反映的都是图像的整体特征，而无法体现图像中所包含的对象或物体。事实上，图像中的空间关系同样是图像检索中非常重要的特征。

所谓空间关系，是指图像中分割出来的多个目标之间的相互的空间位置或相对方向关系，这些关系又可分为连接/邻接关系、交叠/重叠关系和包含/包容关系等。通常空间位置信息可以分为两类：相对空间位置信息和绝对空间位置信息。前一种关系强调的是目标之间的相对情况，如上、下、左、右关系等，后一种关系强调的是目标之间的距离大小以及方位。显而易见，由绝对空间位置可推出相对空间位置，但表达相对空间位置信息相对比较简单。

提取图像空间关系特征有两种方法：一是首先对图像进行自动分割，划分出图像中所包含的对象或颜色区域，然后根据这些区域对图像索引；另一种方法则简单地将图像均匀划分为若干规则子块，对每个图像子块提取特征建立索引。

#### 5. 图像特征提取性能评价

以上几节介绍了多种图像特征的提取，下面对这些图像特征提取进行性能评价。

1) 颜色特征：颜色特征是图像的基本特征之一，同时也是图像检索中应用最为广泛的视觉特征。与其他视觉特征相比，颜色特征对图像本身的尺寸、方向和视角的依赖性较小。颜色特征的提取也相对容易，因而基于颜色特征的图像检索受到了人们广泛的重视和研究。但是仅使用颜色特征查询时，如果数据库很大，常会将许多不需要的图像也检索出来。

2) 纹理特征：在检索具有粗细、疏密等方面较大差别的纹理图像时，利用纹理特征是一种有效的方法。但当纹理之间的粗细、疏密等易于分辨的信息之间相差不大时，通常的纹理特征很难准确地反映出人的视觉感觉不同的纹理之间的差别。




而且纹理特征一个很明显的缺点是当图像的分辨率变化时,所计算出来的纹理可能会有较大偏差。另外,还可能受到光照、反射情况的影响。

3) 形状特征:要把一个图像中所包含的各个物体区分开来,是一件困难的事,而对于人眼来说,则是一件很容易的事情。这是由于图像处理技术,还没有达到人眼视觉的区别程度,因此在利用形状特征进行检索时,必然会出现截然相反的结果。

4) 空间关系特征:空间关系特征的使用可加强对图像内容的描述区分能力,但空间关系特征常对图像或目标的旋转、反转和尺度变化等比较敏感。另外,在实际应用中,仅仅利用空间信息往往是不够的,不能有效准确地表达场景信息。为了检索,除使用空间关系特征外,还需要其他特征来配合。

从颜色、纹理、形状和空间关系特征的特点可以看出,仅基于一种特征的方法只能表达图像的部分属性,由于对图像内容的描述比较片面,缺少足够的区分信息,在图像有较大变化(如尺度或方向)的场合常不能取得理想的检索效果。综合图像的多种特征(颜色、纹理、形状和空间关系等)进行综合检索的技术就是为了克服单一特征检索存在的局限提出来的,从而提高图像检索的查准率与查全率。

 本章中的图像检索系统就是综合了颜色和形状特征的多特征图像检索系统。

### 8.1.2 相似度计算技术

颜色、纹理和形状等图像特征被提取出来后,形成了特征向量,这时就可以用特征向量来表征对应的图像。图像检索的关键就在于判断待检索图像同数据库中图像间的相似度,也就是确定检索图像同数据库图像特征向量间的距离。

显然,一个合适的特征相似度计算方法对图像检索的速度和精度结果影响很大。理想的相似度计算方法应该满足人的视觉特性,也就是说视觉上相似的图像间应具有较小的距离,而视觉上不相似的图像间应具有较大的距离。目前在图像检索中常用到的主要相似度计算方法都是基于向量空间模型的,将视觉特征看做向量空间中的某个点,通过计算两个点之间的距离来度量图像特征间的相似度。下面对常用的相似度计算方法分别予以介绍:

1) 欧氏距离。目前,大多数的图像检索系统的相似度计算是基于欧氏距离函数测度的,欧氏距离是最常见的距离度量函数,日常理解的距离就可以用欧氏距离来度量。欧氏距离具体定义如下:

$$d(A, B) = \left[ \sum_{i=1}^n |a_i - b_i|^2 \right]^{1/2} \quad (8-29)$$

欧氏距离的优点是简便易行,有清晰的物理意义,其计算复杂度较小,具有空间旋转不变性的特点。但是,由于采用欧氏距离时事先假定了图像特征的各分量之间是正交无关,而且各维数的重要程度相同,因而无法有效地模仿人类对视觉内容的所有感知,所以,必要时需要采用其他

的相似度计算方法来进行特征相似度匹配与索引。

2) 马氏距离。如果特征向量的各个分量间具有相关性或者具有不同权重时,可以采用马氏距离,其定义为:

$$d(A, B) = (A - B)^T C^{-1} (A - B) \quad (8-30)$$

其中  $C$  表示特征向量协方差矩阵。

如果各个特征向量相互独立,马氏距离还可以进一步简化,因为这时只需要计算每个分量的方差  $c_i$ 。简化后的马氏距离如下式:

$$d(A, B) = \sum_{i=1}^N \frac{(a_i - b_i)^2}{c_i} \quad (8-31)$$

3) 直方图相交法。直方图相交法是由 Swain 等人于 1991 年首次提出的,直方图相交法能较好地抑制背景的影响,但计算量偏大。假设  $A$  和  $B$  是含有  $n$  个 bin 的颜色直方图,直方图相交是指每个 bin 中共有的像素数量。其数学描述为:

$$d(A, B) = \sum_{i=1}^n \min(a_i, b_i) \quad (8-32)$$


4) 二次式距离。

二次式距离比欧氏距离和直方图相交法更为有效。相对于前两者,二次式距离由于考虑了不同颜色之间存在的相似度及颜色之间的相关性,检索结果更加符合人的视觉感觉,但相关性对称矩阵的计算量较大。二次式距离的数学描述为:

$$d(A, B) = (A - B)^T M (A - B) \quad (8-33)$$

其中  $M = [m_{ij}]$ ,  $m_{ij}$  表示直方图中下标为  $i$  和  $j$  的两种颜色之间的相似度。颜色相似性矩阵  $M$  从对色彩心理学的研究中获得。

---

 本章中的图像检索系统采用欧氏距离计算图像特征的相似度。

---

## 8.2 系统功能

本章设计实现的是一个基于内容的图像检索系统,主要功能就是利用各种图像特征从图像库中检索出与待检图像相似的图像。

### 8.2.1 功能描述

该系统主要实现以下功能:

- 1) 在基于颜色特征模式下,利用颜色直方图、累计直方图以及颜色矩进行图像检索。
- 2) 在基于形状特征模式下,利用不变矩方法进行图像检索。

3) 在基于颜色和形状综合特征模式下, 利用颜色、形状所占权值得到综合特征, 进行图像检索。

## 8.2.2 界面效果

运行程序后的初始界面效果如图 8-1 所示, 系统设置后的效果如图 8-2 所示, 图 8-3 展示了基于颜色特征下的检索结果, 图 8-4 展示了基于形状特征下的检索结果, 图 8-5 展示了基于颜色和形状综合特征下的检索结果。

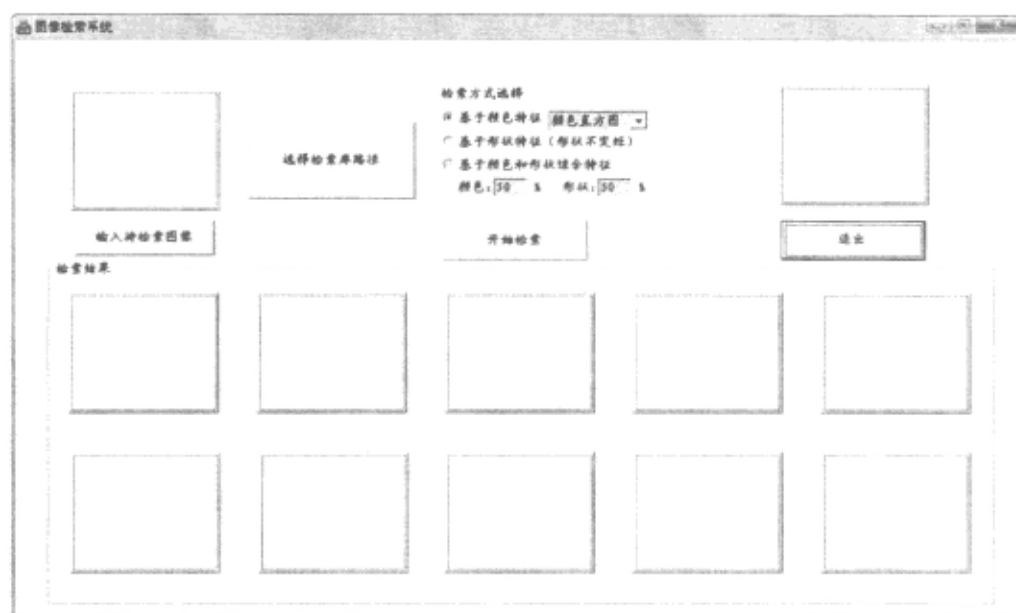


图 8-1 初始界面效果图



图 8-2 系统设置后的效果图



图 8-3 基于颜色特征的检索结果



图 8-4 基于形状特征的检索结果



图 8-5 基于颜色和形状综合特征的检索结果

## 8.3 系统结构与流程

图像检索系统包括对话框显示模块、系统设置模块和图像检索模块。本节主要介绍系统的总体结构和主要流程。

### 8.3.1 总体结构

该系统的主要程序结构模块如图 8-6 所示。

### 8.3.2 主要流程

本章的图像检索系统的主要流程如图 8-7 所示。

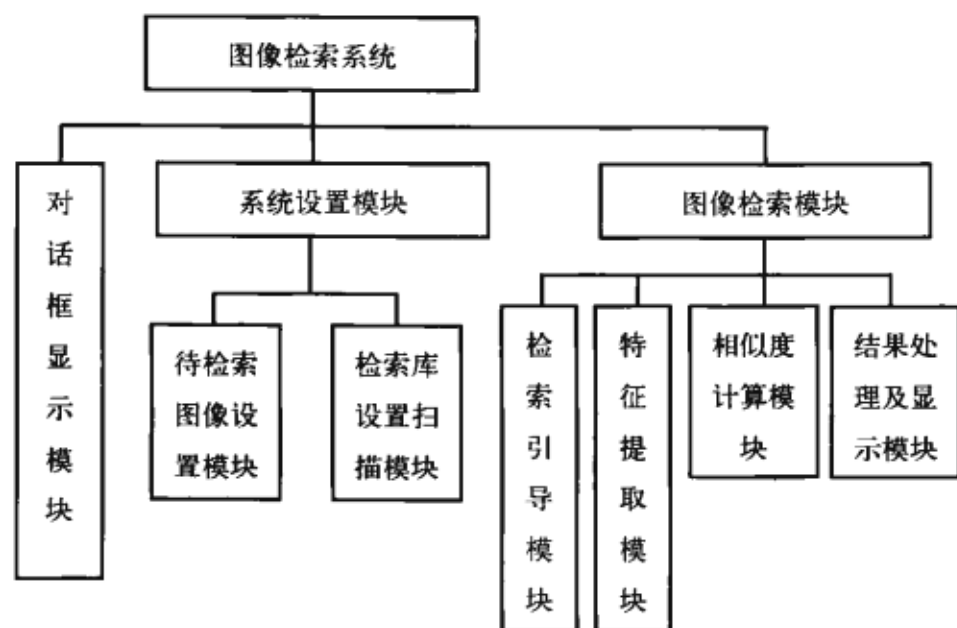


图 8-6 图像检索系统总体结构

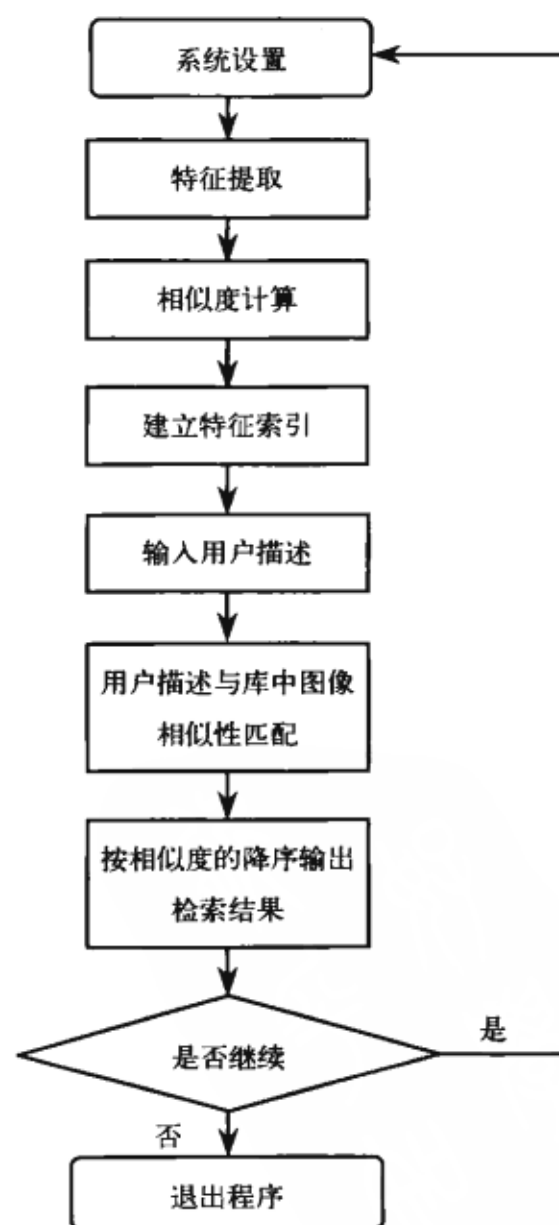


图 8-7 图像检索系统主要流程图

## 8.4 编程实现

图像检索系统采用 VC 2008 开发平台编程实现。

### 8.4.1 系统设置模块

打开图像检索系统后,首先进入系统设置模块。本模块将提供用户设置待检索图像、检索库路径的接口,同时将待检索图像路径和检索库路径分别保存在 CString strfile 和 CString m\_strPath 全局变量中,最后还会对检索库进行图像扫描。本模块共包括两部分:待检索图像设置模块和检索库设置扫描模块。

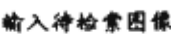
本模块中使用了如下标志位。

- bool open\_pic: 设置待检索图像标志位, true 表示待检索图像已经设置成功。
- bool dir: 设置检索库标志位, true 表示检索库已经设置成功。

本模块中有如下常用全局变量。

- CString strfile: 待检索图像的路径。
- CString m\_strPath: 检索库路径。
- int tempi: 临时的检索库图像计数器。
- int counts: 检索库图像计数器。
- CString\* temp[100]: 检索库中图像路径。

#### 1. 待检索图像设置模块

待检索图像的设置和显示主要是通过单击  按钮进入函数 CImagetrievalDlg::OnOpen()实现的。具体代码如下:

```
void CImagetrievalDlg::OnOpen()
{
    CFileDialog fileOpenDlg(TRUE);
    if (fileOpenDlg.DoModal () != IDOK) return;
    open_pic=true;           //标志位设置为 true, 表示待检索图像已设置
    CWnd* pWnd = GetDlgItem(IDC_VIEW);
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();
    POSITION pos = fileOpenDlg.GetStartPosition(); //得到文件位置
    strfile = fileOpenDlg.GetNextPathName(pos); //得到待检索图像的路径
    ShowPic(strfile, IDC_VIEW);           //显示待检索图像
}
```

其中调用了 CImagetrievalDlg::ShowPic()函数实现图像的显示功能。函数中的参数 pathfile 为待



显示图像的路径, idc 为图像显示控件的 ID 号, 此函数在其他模块中也会被调用到, 其代码如下:

```
void CImagetrievalDlg::ShowPic(CString pathfile, int idc)
{
    CBitmap hbmp;
    HBITMAP hbitmap;
    //将 pStatic 指向要显示的地方
    CStatic *pStaic;
    pStaic=(CStatic*)GetDlgItem(idc);
    //装载资源
    hbitmap=(HBITMAP)::LoadImage (/*::AfxGetInstanceHandle()*/NULL,pathfile,
        IMAGE_BITMAP, 0,0,LR_LOADFROMFILE|LR_CREATEDIBSECTION);
    hbmp.Attach(hbitmap);
    //获取图像格式
    BITMAP bm;
    hbmp.GetBitmap(&bm);
    //创建临时的内存 DC 对象
    CDC dcMem;
    dcMem.CreateCompatibleDC(pStaic->GetDC());
    CBitmap *poldBitmap=(CBitmap*)dcMem.SelectObject(hbmp);
    CRect lRect;
    pStaic->GetClientRect(&lRect);
    lRect.NormalizeRect();
    //显示位图
    pStaic->GetDC()->StretchBlt(lRect.left ,lRect.top ,lRect.Width(),lRect.Height(),
    ght(),
        &dcMem,0 ,0,bm.bmWidth,bm.bmHeight,SRCCOPY);
    dcMem.SelectObject(&poldBitmap);
    pStaic->ReleaseDC(&dcMem);
}
```

## 2. 检索库设置扫描模块

检索库设置扫描模块的主要功能是提供用户设置检索库的接口, 然后系统会对检索库进行扫描, 得到检索库中图像的个数及各个图像的具体路径。其中检索库中图像个数存入 counts 中, 各个图像的具体路径存入 temp[100]中, 以便在以后的模块中调用。本模块是通过单击 选择检索库路径 按钮进入函数 CImagetrievalDlg::OnPath()实现的。具体代码如下:

```
void CImagetrievalDlg::OnPath()
{
    //打开通用对话框, BROWSEINFO 结构中包含了用户选中目录的重要信息
    BROWSEINFO browse;
    ZeroMemory(&browse,sizeof(browse)); //fills a block of memory with zeros.
    browse.hwndOwner = NULL;
    browse.pszDisplayName = m_strPath.GetBuffer(MAX_PATH);
    browse.lpszTitle = "请选择一个图像目录";
    //SHBrowseForFolder 函数返回 ITEMIDLIST 指针, 包含了用户选择文件夹的信息
```

```

LPITEMIDLIST lpItem = SHBrowseForFolder(&browse);
if(lpItem == NULL) return;
m_strPath.ReleaseBuffer();
//SHGetPathFromIDList 把项目标志符列表转换为文档系统路径
if(SHGetPathFromIDList(lpItem,m_strPath.GetBuffer(MAX_PATH)) == false) return;
m_strPath.ReleaseBuffer();
dir=true; //标志位设置为 true, 表示待检索图像已设置
AfxMessageBox("您选择的目录为:"+m_strPath,MB_ICONINFORMATION|MB_OK);
//扫描检索库
CString tempPath;
CString temps;
tempPath=m_strPath;
tempPath.TrimRight();tempPath.TrimLeft();
CString strfilepath=tempPath;
tempI=0;
counts=0; //计数器清零
//检索库中图像个数放入 counts 中, 其路径放入 temp[100] 中
StartDir(strfilepath);
temps.Format("该目录下共有%d 幅图像!",counts);
AfxMessageBox(temps,MB_ICONINFORMATION|MB_OK);
}

```

从以上代码可以看出, 检索库的扫描功能是通过调用 CImagetrievalDlg::StartDir(), 以及在 CImagetrievalDlg::StartDir()中进一步调用 CImagetrievalDlg::RunDir()实现的。CImagetrievalDlg::StartDir()首先通过调用 CImagetrievalDlg::RunDir()扫描当前目录下的图像 (不包括子目录), 若发现子目录则递归调用 CImagetrievalDlg::StartDir()查找子目录下的图像, 直到目录中再无子目录为止。其具体代码如下:

```

void CImagetrievalDlg::StartDir(const CString &strfile1)
{
    BOOL yesno;
    CFileFind find;
    char tempFileFind[200];
    sprintf(tempFileFind,"%s\\*.*",strfile1);
    RunDir(strfile1); //在当前目录中查找图像, 不搜索子目录
    yesno = (BOOL)find.FindFile(tempFileFind);
    //查找当前目录下子目录中的文件
    while(yesno)
    {
        yesno = find.FindNextFile();
        if (find.IsDots() != TRUE) //过滤缺省目录
        {
            char foundFileName[200];
            strcpy(foundFileName,find.GetFileName().GetBuffer(200));
            if((find.IsDirectory() == TRUE)) //判断是否为目录
            {

```

```

        char tempDir[200];
        sprintf(tempDir, "%s\\%s", strfile1, foundFileName); //获得子目录路径
        StartDir(tempDir);          //递归调用, 查找子目录中的图像
    }
}
find.Close();
return;
}
void CImagetrievalDlg::RunDir(const CString &strfile2)
{
    BOOL yesno;
    CFileFind find;
    char tempFileFind[200];
    sprintf(tempFileFind, "%s\\*.bmp", strfile2);
    yesno = find.FindFile(tempFileFind); //在当前目录下查找 BMP 文件
    while(yesno)
    {
        yesno = find.FindNextFile();
        char foundFileName[200];          //临时存储查找到的图像名
        strcpy(foundFileName, find.GetFileName().GetBuffer(200)); //获取图像名
        if(!find.IsDots())                //过滤缺省目录
        {
            char tempFileName[200];
            sprintf(tempFileName, "%s\\%s", strfile2, foundFileName);
            CString strfilepath1;
            strfilepath1.Format("%s", tempFileName); //获取图像完整路径
            counts++;
            temp[tempi] = new CString(strfilepath1); //保存图像完整路径
            tempi++;
        }
    }
    find.Close();
    return;
}

```

#### 8.4.2 图像检索模块

此模块为本系统的核心部分, 主要涉及图像特征提取技术和图像相似度计算技术。本系统中共有 3 种图像特征提取模式, 即基于颜色特征模式、基于形状特征模式以及基于颜色和形状综合特征模式, 而基于颜色特征模式中还包括 3 种方法: 颜色直方图、累计直方图以及颜色矩。另外本系统中统一使用欧氏距离计算图像相似度, 虽然方法相同, 但是在不同的特征提取模式下, 图像相似度计算的对象是不同的。

为了减少系统的复杂度, 需要建立一个图像检索方法引导模块, 引导系统在不同的图像特征

提取模式下调用不同的特征提取函数和相似度计算函数，最后进入结果处理及显示模块。故而，图像检索模块共包括 4 个子模块，分别为：检索引导模块、特征提取模块、相似度计算模块以及结果处理及显示模块。

图像检索模块中使用了两个重要标志位。

- bool color: 基于颜色特征方法计算状态标志位。若值为 true 则表示此方法在当前检索库、待检索图像下计算过。
- bool shape: 基于形状特征方法计算状态标志位。若值为 true 则表示此方法在当前检索库、待检索图像下计算过。

图像检索模块中使用了类 picture，它的对象用来存储不同方法下检索库的相关检索信息。

```
class picture
{
public:
    CString tp[100];           //图像路径
    double num[100];           //图像与待检索图像的距离
    CString strfile_old;
    CString m_strPath_old;
};
```

图像检索模块中有如下常用全局变量。

- int method: 图像特征提取模式，值为 1 表示基于颜色特征模式，值为 2 表示基于形状特征，值为 3 表示基于颜色和形状综合特征模式。
- int c\_method: 基于颜色特征模式下的方法选择，值为 1 表示颜色直方图，值为 2 表示累计直方图，值为 3 表示颜色矩。
- double pix[1000][1000]: 当前分析图像的像素。
- double feature\_shape[8]: 待检索图像的形状特征。
- double feature\_shape\_1[8]: 当前分析图像的形状特征。
- double feature\_color[3][12]: 待检索图像的颜色特征。
- double feature\_color\_1[3][12]: 当前分析图像的颜色特征。
- picture image\_color\_1: 基于颜色特征颜色直方图方法下的检索信息。
- picture image\_color\_2: 基于颜色特征累计直方图方法下的检索信息。
- picture image\_color\_3: 基于颜色特征颜色矩方法下的检索信息。
- picture image\_color\_temp: 临时的基于颜色特征方法下的检索信息。
- picture image\_shape: 基于形状特征方法下的检索信息。
- picture image\_shape\_temp: 临时的基于形状特征方法下的检索信息。
- picture image: 基于颜色和形状综合特征方法下的检索信息。

## 1. 检索引导模块

点击  按钮就进入了检索引导模块, 本模块通过函数 CImagetrievalDlg::OnStart() 实现。其代码如下:


```
void CImagetrievalDlg::OnStart()
{
    //检查是否设置了待检索图像和检索库路径
    if(open_pic==false)
    {
        AfxMessageBox("请先设置待检索图像");
        return;
    }
    if(dir==false)
    {
        AfxMessageBox("请设置检索库路径");
        return;
    }
    if(method==1)
    {
        switch(c_method)
        {
        case 1:
            general(strfile,1);           //提取待检索图像颜色直方图
            break;
        case 2:
            succession(strfile,1);        //提取待检索图像累计直方图
            break;
        case 3:
            centerM(strfile,1);           //提取待检索图像颜色矩
            break;
        }
        Color_SeekImage();                //基于颜色特征的相似度计算函数
    }
    if(method==2)
    {
        torque(strfile,1);               //提取待检索图像的不变矩
        Shape_SeekImage();               //基于形状特征的相似度计算函数
        sort();
    }
    if(method==3)
    {
        torque(strfile,1);
        Shape_SeekImage();
        switch(c_method)
        {
        case 1:
```

```

        general(strfile,1);
        break;
        case 2:
        succession(strfile,1);
        break;
        case 3:
        centerM(strfile,1);
        break;
    }
    Color_SeekImage();
}
}

```

通过以上代码可以看出：在任何特征模式下，本函数都是先调用了特征提取函数，然后调用了相应的相似度计算函数（在形状特征模式下，最后还要调用结果处理及显示函数）。

 由于在基于颜色和形状综合特征模式下需要同时调用颜色特征模式下的相似度计算函数和形状特征模式下的相似度计算函数，因此，为了避免重复计算，在颜色特征模式下的相似度计算函数中调用了结果处理及显示函数，而在形状特征模式下的相似度计算函数中则没有。

## 2. 特征提取模块

特征提取模块中共有 4 个特征提取函数：颜色直方图提取函数、累计直方图提取函数、颜色矩提取函数和不变矩提取函数。下面对这几个函数进行详细介绍。

以下 4 个函数中都有两个传递参数：CString pathfile 和 int mode。pathfile 为图像的路径，mode 为模式状态位，值为 1 时，表示计算的是待检索图像，计算出来的颜色特征需要保存在 feature\_color[3][12] 中，计算出来的形状特征需要保存在 feature\_shape[8] 中；值为 2 时，表示计算的是检索库中的图像，计算出来的颜色特征需要保存在 feature\_color\_1[3][12] 中，计算出来的形状特征需要保存在 feature\_shape\_1[8] 中。

### （1）颜色直方图提取函数：CImagetrievalDlg::general()

根据颜色直方图的计算公式（8-1），其代码如下：

```

void CImagetrievalDlg::general(CString pathfile,int mode)
{
    CBitmap hbmp;
    HBITMAP hbitmap;
    //装载资源
    hbitmap=(HBITMAP)::LoadImage (::AfxGetInstanceHandle(),pathfile,
    IMAGE_BITMAP,0,0,LR_LOADFROMFILE|LR_CREATEDIBSECTION);
    hbmp.Attach(hbitmap);
    //获取图像格式
    BITMAP bm;
    hbmp.GetBitmap(&bm);
}

```



```

//创建临时的内存 DC 对象
CDC dcMem;
dcMem.CreateCompatibleDC(GetDC());
dcMem.SelectObject(hbmp);
int width=bm.bmWidth;
int height=bm.bmHeight;
int totalnum= height *width;
//初始化
long m_graph[3][12];
for(int i=0;i<3;i++)
    for(int j=0;j<12;j++)
    {
        m_graph[i][j]=0;
        if(mode==1)
            feature_color[i][j]=0.0;
        else
            feature_color_1[i][j]=0.0;
    }
COLORREF color;
double h=0,s=0,v=0;
//计算颜色直方图
for(long y=0;y<height;y++)
{
    for(long x=0;x<width;x++)
    {
        color=dcMem.GetPixel(x,y);
        //颜色空间转换
        RGBToHSV(GetRValue(color),GetGValue(color),GetBValue(color),&h,&s,&v);
        int result_h=(int)(6*h/PI);
        if( result_h ==12)
            m_graph[0][11]++;
        else
            m_graph[0][result_h]++;
        int result_s=(int)(s*12);
        if( result_s ==12)
            m_graph[1][11]++;
        else
            m_graph[1][result_s]++;
        int result_v=(int)(v*12);
        if( result_v ==12)
            m_graph[2][11]++;
        else
            m_graph[2][result_v]++;
    }
}
//保存颜色直方图
if(mode==1)

```

```

{
for(int i=0;i<3;i++)
    for(int j=0;j<12;j++)
        feature_color[i][j]=((float)m_graph[i][j])/((float)totalnum);
}
else
{
for(int i=0;i<3;i++)
    for(int j=0;j<12;j++)
        feature_color_1[i][j]=((float)m_graph[i][j])/((float)totalnum);
}
}

```

如前面核心内容中所述,在进行颜色直方图提取之前,首先需要将颜色空间转为 HSV 空间,本函数通过调用 CImagetrievalDlg::RGBToHSV()函数实现此转换,其代码如下:

```

void CImagetrievalDlg::RGBToHSV(int r,int g,int b,double *h,double *s,double *v)
{
    *h=acos((r-g+r-b)/(2.0*sqrtf((float)(r-g)*(r-g)+(float)(r-b)*(g-b))));
    if(b>g)
        *h=2*PI-*h;
    *s=(mymax(r,g,b)-mymin(r,g,b))/(float)mymax(r,g,b);
    *v=mymax(r,g,b)/255.0;
}

```

## (2) 累计直方图提取函数: CImagetrievalDlg::succession()

通过之前核心技术的介绍以及累计直方图公式 (8-3),了解到累计直方图和颜色直方图相似,只在细节处略有不同。因此此处不再赘述,只展示其核心且与颜色直方图不同的部分:

```

//计算累计直方图
for(long y=0;y<height;y++)
{
    for(long x=0;x<width;x++)
    {
        color=dcMem.GetPixel(x,y);
        RGBToHSV(GetRValue(color),GetGValue(color),GetBValue(color),&h,&s,&v);
        int result_h=(int)(6*h/PI);
        if( result_h ==12)
            m_graph[0][11]++;
        else
            m_graph[0][result_h]++;
        int result_s=(int)(s*12);
        if( result_s ==12)
            m_graph[1][11]++;
        else
            m_graph[1][result_s]++;
    }
}

```

```

        int result_v=(int)(v*12);
        if( result_v ==12)
            m_graph[2][11]++;
        else
            m_graph[2][result_v]++;
    }
}

```

### (3) 颜色矩提取函数: CImagetrievalDlg::centerM()

由于这些特征提取函数对图像的预处理相同,因此此处不再赘述,只展示颜色矩提取函数核心部分,根据颜色矩公式(8-4)、公式(8-5)以及公式(8-6),其核心代码如下:

```

int width=bm.bmWidth;
int height=bm.bmHeight;
int totalnum= height *width;
for(long y=0;y<height;y++)
{
    for(long x=0;x<width;x++)
    {
        color=dcMem.GetPixel(x,y);
        RGBToHSV(GetRValue(color),GetGValue(color),GetBValue(color),&h,&s,&v);
        int result_h=(int)(6*h/PI);
        if( result_h ==12)
            m_graph[0][11]++;
        else
            m_graph[0][result_h]++;
        int result_s=(int)(s*12);
        if( result_s ==12)
            m_graph[1][11]++;
        else
            m_graph[1][result_s]++;
        int result_v=(int)(v*12);
        if( result_v ==12)
            m_graph[2][11]++;
        else
            m_graph[2][result_v]++;
    }
}
for(int i=0;i<3;i++)
    for(int j=0;j<12;j++)
        m_graphf[i][j]=((float)m_graph[i][j])/((float)totalnum);
float m1[3],m2[3],m3[3];
for(int i=0;i<3;i++)
{
    m1[i] = 0.0;
    m2[i] = 0.0;
}

```

```

        m3[i] = 0.0;
    }
    for(int i=0;i<3;i++)
        for(int j=0;j<12;j++)
            m1[i] +=m_graphf[i][j]/12;
    for(int i=0;i<3;i++)
        for(int j=0;j<12;j++)
        {
            m2[i] +=((m_graphf[i][j] - m1[i]) * (m_graphf[i][j] - m1[i]))/12;
            m3[i] +=((m_graphf[i][j] - m1[i]) * (m_graphf[i][j] - m1[i])
                    * (m_graphf[i][j] - m1[i]))/12;
        }
    float zz=1/3;
    for(int i=0;i<3;i++)
    {
        m2[i] = sqrtf(m2[i]);
        m3[i] = (float)pow( m3[i], zz );
    }
}

```

#### (4) 不变矩提取函数: CImagetrievalDlg::torque()

本函数包括以下两步。

##### 1) 对图像进行基于区域的分割, 本函数中采用基于 OTSU 的阈值分割法。

阈值分割法是一种基于区域的图像分割技术, 其基本原理是: 通过设定不同的特征阈值, 把图像像素点分为若干类。例如: 大于阈值的像素变为 0, 小于阈值的像素变为 1, 将图像二值化。因此, 阈值分割的关键在于阈值的选取。

而基于 OTSU 的阈值分割法就是利用 OTSU 法计算最佳分割阈值, 然后用这个最佳分割阈值对图像进行阈值分割。

OTSU 法也叫大津法, 是大津于 1979 年提出的。OTSU 法中有一个重要的式子:

$$g = w_0 \times (u_0 - u)^2 + w_1 \times (u_1 - u)^2$$

从最小灰度值到最大灰度值遍历阈值  $t$ , 当利用  $t$  进行阈值分割使得  $g$  值最大时,  $t$  即为分割的最佳阈值。

该式实际上就是类间方差值, 阈值  $t$  分割出的前景和背景两部分构成了整幅图像, 而前景平均灰度值为  $u_0$ , 像素所占比例为  $w_0$ , 背景平均灰度值为  $u_1$ , 像素所占比例为  $w_1$ , 总平均灰度值为  $u$ , 根据方差的定义即得该式。因方差是灰度分布均匀性的一种度量, 方差值越大, 说明构成图像的两部分差别越大, 当部分目标错分为背景或部分背景错分为目标都会导致两部分差别变小, 因此使类间方差最大的分割意味着错分概率最小。

为了减小计算量, 此处采用等价式:

$$g = w_0 \times w_1 \times (u_0 - u_1)^2$$

找到最佳分割阈值后,即可对图像进行阈值分割。其核心代码如下:

```

COLORREF clr;
for(int x=0;x<width;x++)
{
    for(int y=0;y<height;y++)
    {
        //该函数检索指定坐标点的像素的 RGB 颜色值
        clr=dcMem.GetPixel(x,y);
        R=GetRValue(clr);
        G=GetGValue(clr);
        B=GetBValue(clr);
        pix[x][y]=0.299*R+0.587*G+0.114*B;
    }
}
//大津法求阈值
int index_max;                //最佳阈值
double g_max;
g_max=0;
for(int index=0;index<256;index++)
{
    double c0,c1,sum_0,sum_1;
    double w0,w1,u0,u1,g;
    c0=c1=sum_0=sum_1=w0=w1=u0=u1=g=0;
    for(int x=0;x<width;x++)
    {
        for(int y=0;y<height;y++)
        {
            if(pix[x][y]<index)
            {
                sum_0=sum_0+pix[x][y]; //统计背景灰度值
                c0++;                  //统计背景像素值
            }
            else
            {
                sum_1=sum_1+pix[x][y]; //统计前景灰度值
                c1++;                  //统计前景像素值
            }
        }
    }
    w0=c0/(c0+c1);                  //背景像素所占比例
    w1=c1/(c0+c1);                  //前景像素所占比例
    if(c0==0)
        u0=0;
    else
        u0=sum_0/c0;                //背景平均灰度值
    if(c1==0)

```

```

        ul=0;
    else
        ul=sum_1/c1;                //前景平均灰度值
    g=w0*w1*(u0-ul)*(u0-ul);
    if(g>g_max)                      //保存最佳阈值
    {
        g_max=g;
        index_max=index;
    }
}
//根据所求阈值进行阈值分割
for(int x=0;x<width;x++)
{
    for(int y=0;y<height;y++)
    {
        if(pix[x][y]<index_max)
            pix[x][y]=0;
        else
            pix[x][y]=1;
    }
}

```

## 2) 对分割后的区域进行不变矩提取。

这一步根据不变矩公式(8-28)即可得到。函数核心代码如下:

```

//0+0 阶矩
double m00=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m00+=pix[x][y];
//1+0 阶矩
double m10=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m10+=(x+1)*pix[x][y];
//0+1 阶矩
double m01=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m01+=(y+1)*pix[x][y];
//1+1 阶矩
double m11=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m11+=(y+1)*(x+1)*pix[x][y];
//2+0 阶矩
double m20=0;

```



```

for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m20+=(x+1)*(x+1)*pix[x][y];
//0+2 阶矩
double m02=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m02+=(y+1)*(y+1)*pix[x][y];
//3+0 阶矩
double m30=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m30+=(x+1)*(x+1)*(x+1)*pix[x][y];
//0+3 阶矩
double m03=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m03+=(y+1)*(y+1)*(y+1)*pix[x][y];
//1+2 阶矩
double m12=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m12+=(x+1)*(y+1)*(y+1)*pix[x][y];
//2+1 阶矩
double m21=0;
for(int x=0;x<width;x++)
    for(int y=0;y<height;y++)
        m21+=(x+1)*(x+1)*(y+1)*pix[x][y];
//求图像的区域重心
double xbar,ybar;
xbar=(double)m10/m00;
ybar=(double)m01/m00;
//求中心矩
double eta11,eta20,eta02,eta30,eta03,eta21,eta12;
eta11=(m11-ybar*m10)/(m00*m00);
eta20=(m20-xbar*m10)/(m00*m00);
eta02=(m02-ybar*m01)/(m00*m00);
eta30=(m30-3*xbar*m20+2*xbar*xbar*m10)/(m00*m00*sqrt(m00));
eta03=(m03-3*ybar*m02+2*ybar*ybar*m01)/(m00*m00*sqrt(m00));
eta21=(m21-2*xbar*m11-ybar*m20+2*xbar*xbar*m01)/(m00*m00*sqrt(m00));
eta12=(m12-2*ybar*m11-xbar*m02+2*ybar*ybar*m10)/(m00*m00*sqrt(m00));
//HU 不变矩
double phi[8];
phi[0]=eta20+eta02;
phi[1]=(eta20-eta02)*(eta20+eta02)+(4*eta11*eta11);
phi[2]=((eta30-3*eta12)*(eta30-3*eta12))+((3*eta21-eta03)*(3*eta21-eta03));
phi[3]=((eta30+eta12)*(eta30+eta12))+((eta21+eta03)*(eta21+eta03));

```

```

phi[4]=(eta30-3*eta12)*(eta30+eta12)*((eta30+eta12)*(eta30+eta12)-3*(eta21+eta03)*
(eta21+eta03)+(3*eta21-eta03)*(eta21+eta03)*
(3*(eta30+eta12)*(eta30+eta12)-(eta21+eta03)*(eta21+eta03)));
phi[5]=(eta20-eta02)*((eta30+eta12)*(eta30+eta12)-(eta21+eta03)*(eta21+eta03))+
4*eta11*(eta30+eta12)*(eta21+eta03);
phi[6]=(3*eta21-eta03)*(eta30+eta12)*((eta30+eta12)*(eta30+eta12)-3*(eta21+eta03)*
(eta21+eta03)+(3*eta12-eta30)*(eta21+eta03)*
(3*(eta30+eta12)*(eta30+eta12)-(eta21+eta03)*(eta21+eta03)));
//图像离心率
phi[7]=((eta20-eta02)*(eta20-eta02)+4*eta11*eta11)/((eta20+eta02)*(eta20+eta02));
if(mode==1)
    for(int i=0;i<8;i++)
    {
        phi[i]=fabs(log(fabs(phi[i])));
        feature_shape[i]=phi[i];
    }
else
    for(int i=0;i<8;i++)
    {
        phi[i]=fabs(log(fabs(phi[i])));
        feature_shape_1[i]=phi[i];
    }

```

### 3. 相似度计算模块

本模块中共有两个相似度计算函数: CImagetrievalDlg::Shape\_SeekImage() 和 CImagetrievalDlg::Color\_SeekImage()。它们都是逐一对检索库中的图像进行分析处理的, 具体处理包括以下几步:

- 1) 提取当前分析图像的特征向量。
- 2) 计算当前分析图像与待检索图像的相似度。
- 3) 保存当前分析图像的相关检索信息。
- 4) 在控件中显示当前正在分析处理的图像。

本模块为整个图像检索系统中计算量最大的部分, 因此为了减小系统的计算时间复杂度, 可以从本模块入手: 根据不同的检索方法, 将相似度计算结果保存在相应的 picture 类文件中, 在待检索图像、检索库都不变的情况下, 若再次调用计算过的检索方法, 则不用重复计算, 直接从相应的 picture 类文件中读取即可。

为了标志在当前的检索库下的当前待检索图像是否已经用当前方法计算过, 本模块使用了如下方法: 将相应 picture 类文件中的 strfile\_old、m\_strPath\_old 和全局变量 strfile、m\_strPath 进行对比, 若不一致, 则此方法并未计算过, 则将标志位 color 或 shape 设置为 0。

由于基于颜色特征的相似度计算函数更复杂一些, 所以这里只介绍基于颜色特征的相似度计算函数, 其代码如下。

```

void CImagetrievalDlg::Color_SeekImage()
{
    for(int count=0; count<counts; count++)
        temp_1[count] = *temp[count];
    switch(c_method)
    {
    case 1:
        if(image_color_1.strfile_old==strfile&&image_color_1.m_strPath_old==
            m_strPath)
            color=1;        //此检索库下的此检索图像已经用此方法计算过
        else
            color=0;
        break;
    case 2:
        if(image_color_2.strfile_old==strfile&&image_color_2.m_strPath_old==
            m_strPath)
            color=1;        //此检索库下的此检索图像已经用此方法计算过
        else
            color=0;
        break;
    case 3:
        if(image_color_3.strfile_old==strfile&&image_color_3.m_strPath_old==
            m_strPath)
            color=1;        //此检索库下的此检索图像已经用此方法计算过
        else
            color=0;
        break;
    }
    for(int pic=0; pic<counts; pic++)//逐个分析检索库中图像
    {
        if(color==0)
        {
            switch(c_method)
            {
            case 1:
                general(temp_1[pic],2);//提取检索的第N幅图的颜色直方图
                break;
            case 2:
                succession(temp_1[pic],2);//提取检索的第N幅图的累计直方图
                break;
            case 3:
                centerM(temp_1[pic],2);//提取检索的第N幅图的颜色矩
                break;
            }
            double dis=0;        //第N幅图与待检索图像的距离
            double dis_temp=0;
            //计算正在检索的第N幅图与待检索图像的距离

```

```

        for(int i=0;i<3;i++)
        {
            for(int j=0;j<12;j++)
                dis_temp+=(feature_color_1[i][j]-feature_color[i][j])*
                    (feature_color_1[i][j]-feature_color[i][j]);
            dis+=sqrt((double)dis_temp);
            dis_temp=0;
        }
        image_color_temp.tp[pic]=temp_1[pic]; //保存第N幅图路径
        image_color_temp.num[pic]=dis; //保存第N幅图与待检索图像的距离
    }
    ShowPic(temp_1[pic], IDC_VIEW1);
}
switch(c_method)
{
case 1:
    if(color==0) //若本方法为第一次计算, 则保存相似度计算结果
    {
        image_color_1=image_color_temp;
        sort();
    }
    else //若本方法已经计算过, 则读取相似度计算结果
    {
        image_color_temp=image_color_1;
        sort();
    }
    //检索过的待检索图像及检索库路径更新
    image_color_1.m_strPath_old=image_color_temp.m_strPath_old=m_strPath;
    image_color_1.strfile_old=image_color_temp.strfile_old=strfile;
    break;
case 2:
    if(color==0)
    {
        image_color_2=image_color_temp;
        sort();
    }
    else
    {
        image_color_temp=image_color_2;
        sort();
    }
    image_color_2.m_strPath_old=image_color_temp.m_strPath_old=m_strPath;
    image_color_2.strfile_old=image_color_temp.strfile_old=strfile;
    break;
case 3:
    if(color==0)
    {

```

```

        image_color_3=image_color_temp;
        sort();
    }
    else
    {
        image_color_temp=image_color_3;
        sort();
    }
    image_color_3.m_strPath_old=image_color_temp.m_strPath_old=m_strPath;
    image_color_3.strfile_old=image_color_temp.m_strPath_old=strfile;
    break;
}
}

```

#### 4. 结果处理及显示模块

通过以上模块的处理，系统得到了相关检索信息 picture 类文件，利用此文件即可调用 CImagetrievalDlg::sort() 函数进行结果处理及显示。其中，基于颜色特征和基于形状特征的结果处理只需要进行相似度排序，保留前 10 个结果即可，而基于颜色和形状综合特征的结果处理则先需要根据颜色、形状特征所占权值求出综合相似度，然后再对综合相似度进行排序处理。其代码如下：

```

void CImagetrievalDlg::sort()
{
    int picnum=1009;           //图像显示控件的 ID 号
    if(method==1)
    {
        CString tp;           //图像路径
        double num;           //图像与待检索图像的距离
        //图像相似度排序
        for(int i=0;i<counts;i++)
        {
            for(int j=i+1;j<counts;j++)
            {
                if(image_color_temp.num[i]>image_color_temp.num[j])
                {
                    tp=image_color_temp.tp[i];
                    num=image_color_temp.num[i];
                    image_color_temp.tp[i]=image_color_temp.tp[j];
                    image_color_temp.num[i]=image_color_temp.num[j];
                    image_color_temp.tp[j]=tp;
                    image_color_temp.num[j]=num;
                }
            }
        }
        //显示前 10 个相似度最高的图像
        for(int i=0;i<10&&picnum<=1018;i++)
        {
            ShowPic(image_color_temp.tp[i],picnum);
            picnum++;
        }
    }
}

```

```

    }
}
if(method==2)
{
    CString tp;           //图像路径
    double num;           //图像与待检索图像的距离
    //图像相似度排序
    for(int i=0;i<counts;i++)
    {
        for(int j=i+1;j<counts;j++)
        {
            if(image_shape_temp.num[i]>image_shape_temp.num[j])
            {
                tp=image_shape_temp.tp[i];
                num=image_shape_temp.num[i];
                image_shape_temp.tp[i]=image_shape_temp.tp[j];
                image_shape_temp.num[i]=image_shape_temp.num[j];
                image_shape_temp.tp[j]=tp;
                image_shape_temp.num[j]=num;
            }
        }
    }
    //显示前10个相似度最高的图像
    for(int i=0;i<10&&picnum<=1018;i++)
    {
        ShowPic(image_shape_temp.tp[i],picnum);
        picnum++;
    }
}
if(method==3)
{
    CString tp;           //图像路径
    double num;           //图像与待检索图像的距离
    //图像相似度排序
    for(int i=0;i<counts;i++)    //综合特征相似度
    {
        image.num[i]=(c_per*image_color_temp.num[i]+
            s_per*image_shape_temp.num[i])/100;
        image.tp[i]=image_color_temp.tp[i];
    }
    for(int i=0;i<counts;i++)
    {
        for(int j=i+1;j<counts;j++)
        {
            if(image.num[i]>image.num[j])
            {
                tp=image.tp[i];
                num=image.num[i];
                image.tp[i]=image.tp[j];

```



```

        image.num[i]=image.num[j];
        image.tp[j]=tp;
        image.num[j]=num;
    }
}
//显示前10个相似度最高的图像
for(int i=0;i<10&&picnum<=1018;i++)
{
    ShowPic(image.tp[i],picnum);
    picnum++;
}
}
}

```

## 8.5 经验分享

1) 关于特征选择与检索精度和速度的问题。为了演示采用不同图像特征进行检索的差异,本章的图像检索系统中列出了几种基于不同特征的检索方式供用户选择。在实际应用中,经常采用的都是多特征融合的方式,以提高检索准确度。但是多特征融合无疑会带来计算量的增加,很可能无法满足一些特殊应用场合对实时性的要求。因此,在实际应用中,要根据具体需求综合考虑精度与速度的问题。

2) 关于查全率和查准率的问题。查全率和查准率是反应检索效果的重要指标。查全率是正确检出的图像占库中全部相似图像的百分比,查准率是正确检出的图像占全部检出图像的百分比。通过设置相似度阈值可以对两个指标进行调控,设定较低的相似度阈值,可以提高查全率,但会降低查准率;设定较高的相似度阈值,可以提高查准率,但会降低查全率。设计检索系统时对此也需要综合考虑。

3) 关于图像索引数据库的问题。本系统没有建立专门的索引数据库来存储图像库中的特征计算结果,因此每次都需要重新计算。如果设计一个索引数据库则能够有效提高检索效率。

## 第9章 细胞检测与计数系统

1984年日本电视连续剧《血疑》在国内播出，山口百惠与三浦友和这对金童玉女演绎的凄美爱情故事牵动了无数中国观众的心。很多人在为女主人公大岛幸子的不幸命运深为悲伤的时候，也把一个晦涩的医学术语“再生障碍性贫血”留在了记忆当中。再生障碍性贫血是一种骨髓造血功能衰竭症，主要表现为骨髓造血功能低下、全血细胞减少、贫血、出血等症状。临床上采用常规检查、骨髓穿刺和骨髓活检等手段进行检测和诊断。其中血常规检查是用针刺法采集指血或耳垂末梢血制成血涂片，然后置于显微镜下观察和计算血细胞数目。本章关注的不是那些让人望而生畏的医学术语，而是如何利用计算机视觉技术自动检测和统计显微镜下血涂片中的细胞。

**本章要点：**

- 显微图像去噪技术
- 颜色空间及其转换技术
- 阈值分割技术
- Blob 分析技术
- 边缘提取技术
- 细胞检测与计数系统的功能描述
- 细胞检测与计数系统的总体结构与主要流程
- 细胞检测与计数系统的编程实现

### 9.1 核心技术原理

细胞检测与计数系统涉及的核心技术主要有显微图像去噪、颜色空间及其转换、阈值分割处理、Blob 分析和边缘提取技术。

#### 9.1.1 显微图像去噪技术


用高性能计算机、高倍放大镜、图像采集卡和摄像头等设备扫描新的血涂片，对高倍放大的血液进行视图操作，然后采集图像将其保存成图像文件，就取得了所需的显微图像。但由于硬件

设施、切片本身或环境条件等因素的影响,在显微镜成像的过程中不可避免的出现噪声,使得图像模糊,信息量不足,给图像后续的处理带来困难,所以在系统中首要的任务就是要去除噪声。

去除噪声的方法有很多,如中值滤波、高斯滤波和均值滤波等。中值滤波是一种非线性滤波技术,对滤除脉冲干扰及图像扫描噪声非常有效,而对随机噪声的抑制较差,时间空间消耗相对较大。高斯滤波是线性滤波,是根据高斯函数的形状来选择权值的,使用起来较为复杂。均值滤波是一种局部空间域处理方法,它用邻近像素的平均值来代替中心点像素的值,这样能明显削弱噪声点,使邻域中灰度接近均匀。虽然均值滤波是以图像模糊为代价来减小噪声的,但它算法简单、计算速度快、效率高是去噪方法中最常用的一种。

本章的细胞检测与计数系统采用的是 $3 \times 3$ 均值滤波。由于原图像是彩色图像,需要对 $R$ 、 $G$ 、 $B$ 三原色通道分别进行平滑。即对一个像素和其八个邻域像素分别求出 $R$ 、 $G$ 、 $B$ 值进行求平均,

然后再分别赋给这个像素的 $R$ 、 $G$ 、 $B$ 。其模板为 $\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$ 。

 图像去噪方法很多,但都各有优缺点,要综合考虑各种因素择取最合适的。

### 9.1.2 颜色空间及其转换技术

本节对数字图像处理中常用颜色空间的特性和应用场景进行探讨,并给出常用颜色空间之间的转换方法。

#### 1. 颜色空间

颜色空间指的是某个三维颜色空间中的一个可见光子集,它包含某个色彩域的所有颜色,颜色空间可用于表示色彩之间的相互关系。比较常用的颜色空间有 $RGB$ 、 $YUV$ 、 $CMY/CMYK$ 、 $YCbCr$ 和 $HSI$ 等。

##### (1) $RGB$ 颜色空间

由于彩色显示器采用红、绿、蓝来生成目标颜色,所以 $RGB$ 颜色空间是计算机图形学最通常的选择,这样可以简化系统的架构与设计。下面用三维的笛卡儿坐标系统来表示 $RGB$ 颜色空间(如图9-1所示)。

##### (2) $YUV$ 颜色空间

$YUV$ 是 $PAL$ 制式和 $SECAM$ 制式采用的颜色空间,其中 $Y$ 代表亮度, $U$ 和 $V$ 代表色度。 $YUV$ 颜色空间的亮度信号 $Y$ 和色度信号 $U$ 、 $V$ 是分离的,只需要用 $U$ 和 $V$ 两个分量即可表示色彩,可以单独编码,易于实现压缩,方便传输和处理,因此它被广泛应用于计算机视频和图像处理之中。

##### (3) $CMY/CMYK$ 颜色空间

$CMY$ 颜色空间一般采用红、绿、蓝的补色青色(Cyan)、品红(Magenta)、黄色(Yellow)

的组合产生新颜色,是一种减色混色系统,常用于彩色打印及彩色印刷中。因为用等量的 CMY 颜色空间中的三种颜色得不到真正的黑色,所以在 CMY 色彩中需要另加一个黑色(K),才能弥补这三个颜色混合不够黑的问题。因此在印刷业中实际上使用的是 CMYK 颜色空间。

#### (4) HSI 颜色空间

HSI 颜色空间反映了人的视觉对色彩的感觉,它用  $H$  (Hue)色调、 $S$  (Saturation)饱和度、 $I$  (Intensity) 亮度三参数描述颜色特性。通常把色调和饱和度统称为色度,用来表示颜色的类别与深浅程度。由于人的视觉对亮度的敏感程度远强于对颜色浓淡的敏感程度,为了便于色彩处理和识别,视觉系统经常采用 HSI 颜色空间,它比 RGB 颜色空间更符合人的视觉特性。HSI 颜色空间还特别适合传统的图像处理函数,如卷积、均化和直方图等,可以通过处理亮度值来实现这些操作。图 9-2 所示为 HSI 双六角锥形的颜色模型。

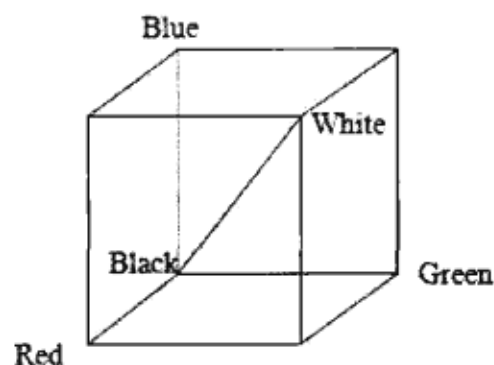


图 9-1 RGB 颜色空间

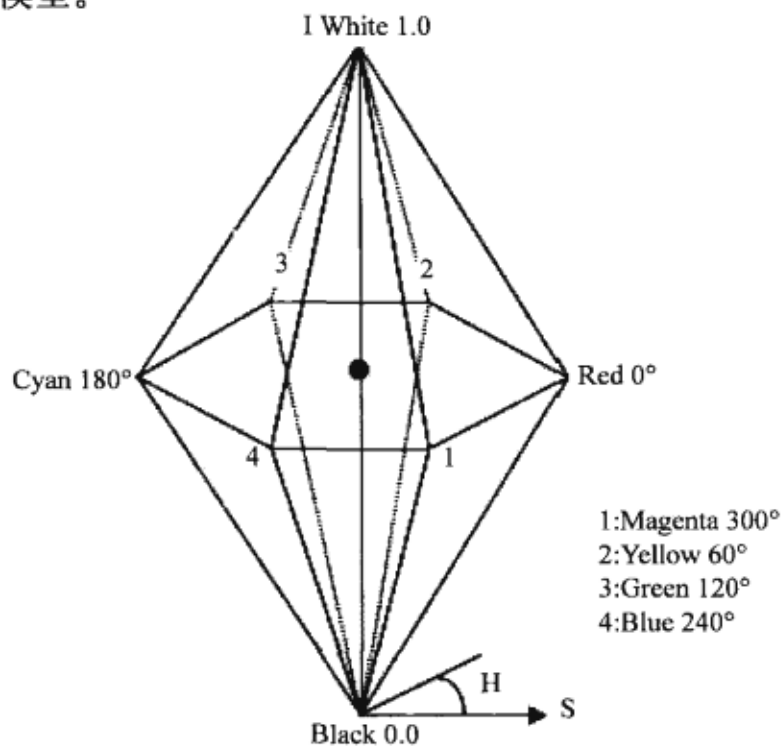


图 9-2 HSI 颜色空间模型

#### (5) YCbCr 颜色空间

YCbCr 是由 YUV 颜色空间衍生而来的,是 YUV 颜色空间的缩放和偏移版本。 $Y$  表示亮度分量,  $Cb$  和  $Cr$  是由  $U$  和  $V$  做调整得到的,  $Cb$  指蓝色色度分量,而  $Cr$  指红色色度分量。YCbCr 被广泛应用在电视显示等领域,也是许多视频压缩编码如 MPEG 和 JPEG 等标准中普遍采用的颜色表示格式。

### 2. 颜色空间转换技术

不同颜色空间适用场景不同,如一幅图像在计算机中用 RGB 显示,用 YUV 或 HSI 编辑处理,

打印输出时要转换成 CMY，印刷时则要转换成 CMYK。因此，在实际应用中，需要在各种不同的颜色空间之间进行转换。下面将详细介绍上述颜色空间之间的转换关系。

(1) RGB 与 CMY/CMYK 之间的转换为：

$$C = 255 - R \quad (9-1)$$

$$M = 255 - G \quad (9-2)$$

$$Y = 255 - B \quad (9-3)$$

在印刷中，为了得到真正的黑色， $K$  值是必不可少的。CMYK 与 RGB 之间的转换关系为：

$$K = \min(C, M, Y) \quad (9-4)$$

$$C = C - K \quad (9-5)$$

$$M = M - K \quad (9-6)$$

$$Y = Y - K \quad (9-7)$$

(2) RGB 与 YUV 之间的转换为：

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.148 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (9-8)$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \quad (9-9)$$

(3) RGB 与 HSI 之间的转换为：

由 RGB 到 HSI 颜色空间转换的几何推导公式为：

$$I = \frac{1}{3}(R + G + B) \quad (9-10)$$

$$S = 1 - \frac{3}{(R + G + B)} [\min(R, G, B)] \quad (9-11)$$

$$H = \arccos \left\{ \frac{\frac{1}{2}[(R - G) + (R - B)]}{[(R - G)^2 + (R - B)(G - B)]^{1/2}} \right\} \quad (9-12)$$

(4) RGB 与 YCbCr 之间的转换为：

$$\begin{bmatrix} Y \\ Cb \\ Cr \\ 1 \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 & 0 \\ -0.1687 & -0.3313 & 0.5000 & 128 \\ 0.5000 & -0.4187 & -0.0813 & 128 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ 1 \end{bmatrix} \quad (9-13)$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1.40200 & 0 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.77200 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb-128 \\ Cr-128 \end{bmatrix} \quad (9-14)$$

除 RGB 之外的颜色空间之间的转换可以由上述公式推导。

### 9.1.3 阈值分割技术

阈值分割就是把一幅图像中像素值处于一定范围内的点赋予特定的值（比如 0 表示黑色），而在这个范围之外的像素点被赋予另外一个值（比如 255 表示白色）或者保持这些像素点的值不变。图像阈值化的目的是要按照灰度级对像素集合进行划分，得到的每个子集形成一个与现实景物相对应的区域，各个区域内部具有一致的属性，而相邻区域布局也有这种一致属性。

阈值法是一种传统的图像分割方法，因其实现简单、计算量小、性能较稳定而成为图像分割方法中最基本和应用最广泛的分割技术，已被应用于很多的领域。例如，在医学应用中，血液细胞图像的分割，磁共振图像的分割。在红外技术应用中，红外无损检测中红外热图像的分割，红外成像跟踪系统中目标的分割。在农业工程应用中，水果品质无损检测过程中水果图像与背景的分割。在这些应用中，分割是对图像进一步分析的前提，分割的准确性将直接影响后续操作的有效性。常用的阈值分割技术有最大方差阈值分割、自适应阈值分割、直方图阈值分割和 HSI 选取阈值分割等。

面向彩色图像处理的颜色模型中最常用的是 HSI 模型，它克服了 RGB 颜色模型不直观以及各分量高度相关的缺点，而且  $H$  和  $S$  分量的大小与人对颜色的感受是紧密相连的，这种彩色描述对人来说是自然的、直观的。其中， $H$ （色调）在 3 个分量中更能反映不同颜色物体的差异，因此对于彩色图像的阈值分割更有效。本系统中，为了从血液图像中提取红细胞，先手动选取一块特征颜色区域，记录特征颜色区域的 HSI 值（实际选出来的为 RGB 颜色空间，通过 `int RgbToHsi(RGB *Rgbhsi, HSI *Hsirgb)` 函数转换为 HSI 颜色空间），然后根据记录的 HSI 值在整个图像内进行搜索，以选出所有红细胞。图 9-3a 为原图像，图 9-3b 为 HSI 阈值分割后的图像。

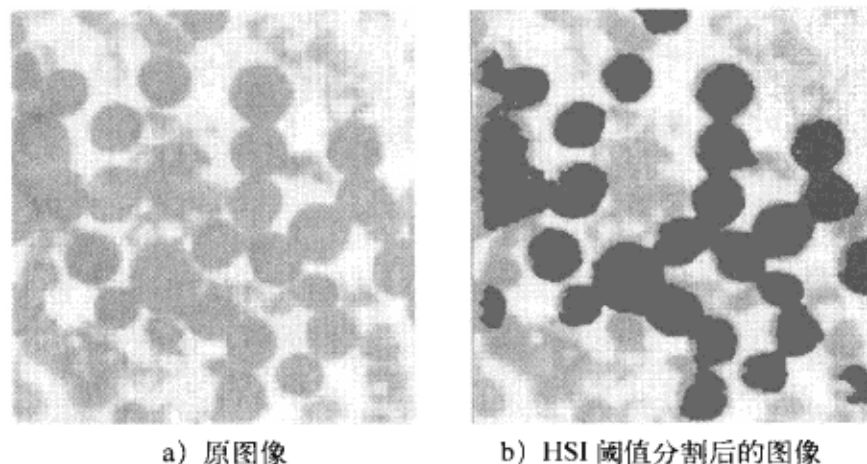


图 9-3 HSI 阈值分割效果图



### 9.1.4 Blob 分析技术

Blob 分析又叫斑点分析,是一种对图像中相同像素的连通域进行分析的技术,该连通域称为 Blob。Blob 算法就是找出图像中的连通域,并确定其数量、大小、面积和位置等信息的一种方法。

Blob 分析适用场合主要有以下几种:

- 对象在尺寸、形状或方向上差异很大(训练模型很难或者不可能)。
- 对象有背景中找不到的截然不同的灰度。
- 对象没有重叠或者接触。

通常的 Blob 分析主要步骤如图 9-4 所示。

在进行阈值分割后,运用连通性规则找到连通区域,目的是提取目标物体。由于是人为选取 HSI 颜色特征区域,可能选取不恰当,或者实际图像处理过程中受光照度不均匀、背景噪声和图像分割算法缺陷等因素的影响,使得图像中存在一些大大小小的孔洞或者碎片,成为伪连通区域,使得提取目标物体不准确,如图 9-5a 所示。这样会使后续操作精确率降低,在腐蚀细胞外围时,内部孔洞也在扩大,所以需要填充孔洞的算法来更新。

填充孔洞的原理是:在阈值处理时,如果像素已被标志,则填充孔洞时,先统计所有连通的非标志区域的面积,这些面积中有相对较大和较小的区域。那些较小或很小的区域往往就是需要填充的孔洞。在算法中,先判断孔洞区域,如果判定某个区域在目标区域范围之内,并且满足其连通部分不与背景区域有相交点,则此区域为孔洞,进行填充。在图像中,被标志区域默认用蓝色标志。填充效果如图 9-5b 所示。

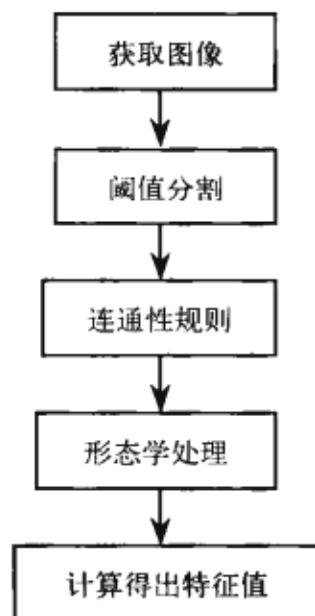


图 9-4 Blob 分析主要步骤

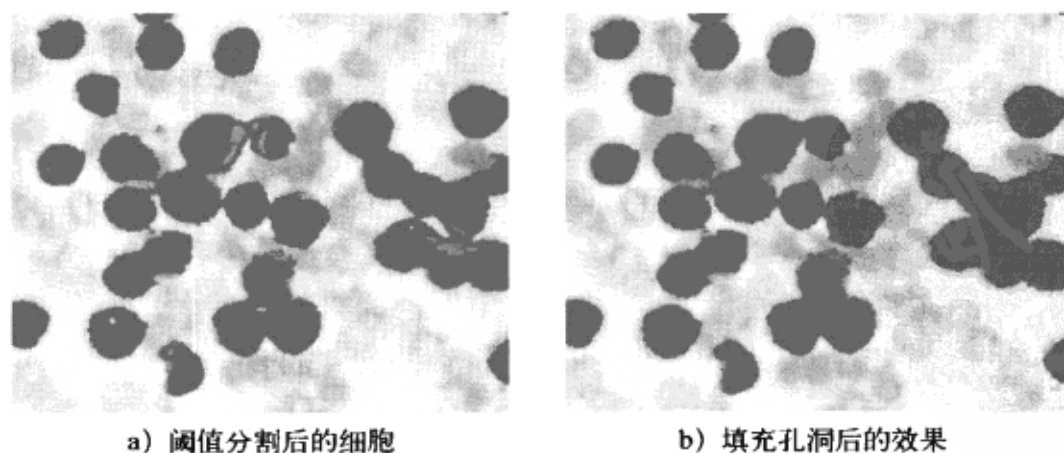


图 9-5 填充孔洞前后对比图

形态学处理是指将数字形态学作为工具从图像中提取对于表达和描绘区域形状有用的图像

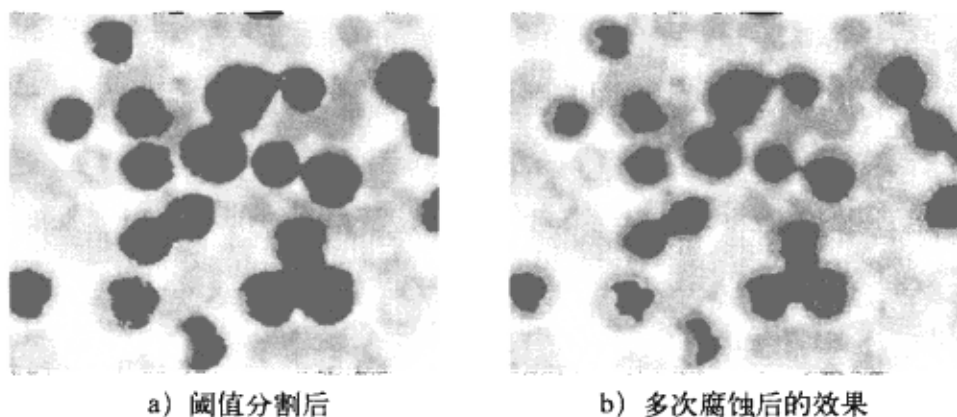
分量,比如边界和骨架。常用的形态学方法有腐蚀、膨胀和细化等。下面结合本章的细胞检测与计数系统来详细介绍这些方法。

### 1. 腐蚀

腐蚀是指从图像边界处消除一些像素,使边界向内部收缩的过程。其主要作用就是消去小且无意义的物体。常用的腐蚀算法主要使用半径为1的结构元素,其模板为  $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$  (十字架形)

和  $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$  (方形)。在实际操作中,一般都会应用连续多次腐蚀,当连续层被消除后,特征对

象将会缩小,直至保留中心点或者消失。在本系统中,在“查找中心点”之前会进行多次腐蚀,对于本系统中的圆形或椭圆形的细胞来说,一次腐蚀或细化半径就会减1,从而获取细胞半径。图9-6所示为多次腐蚀的效果。



a) 阈值分割后

b) 多次腐蚀后的效果

图 9-6 多次腐蚀前后效果对比

### 2. 膨胀

膨胀是腐蚀的对偶运算,可理解为腐蚀背景,使背景合并到目标物体中或连通较近的目标物体。在系统中可以经过多次膨胀使整个红细胞都成为目标区域。

### 3. 细化

细化就是要从原来的图像中去除一些点,但仍要保持原来的形状。经细化之后保留下来的是原图的骨架。所谓骨架,可以理解为图像的中轴,例如孤立点的骨架是它自身;直线的骨架也是它自身;圆的骨架是它的圆心;长方形的骨架是它的长方向上的中轴线;正方形的骨架是它的中心点。在应用中,对图像进行细化有助于突出形状特征和减少冗余的信息。

图像细化运算过程要满足两个条件:

1) 在细化过程中,图像应该是有规律地保持原形基本不变地缩小。

2) 在图像逐渐缩小的过程中, 其连通性应该保持不变。

通常, 细化算法都是根据 8 邻域点的情况进行的, 本系统的具体细化算法为:

1) 如果被标志点的连通区域中有任何一个非边界点, 则去掉边界。否则将所有边界点标志为临时中心点。

2) 边界生成。假如有一个被标志的点的四周有任何一个点未被标志, 则此点为边界点。也可以使用八方向判断, 即假如一个被标志的点的八方向邻接点有任何一个未标志, 则此点为边界点。系统中循环使用四方向和八方向边界。

3) 重复进行步骤 1 和步骤 2, 直到所有点都被访问。

要想得到最终的中心点, 还需要进行一些修正。主要是针对噪声影响和操作 (如细化) 不精确造成的错误, 去除一些伪细胞和对所有被标志为临时中心点并且连通的像素进行坐标平均, 将平均值作为最终的中心点。

修正主要包括以下几个方面:

1) 对于面积较小的区域, 很有可能是噪声造成的。需要对面积很小的区域进行颜色匹配检查, 如果其包含的特征色的面积小于总面积的  $2/3$ , 就认为是噪声造成的 (如图 9-7 所示)。

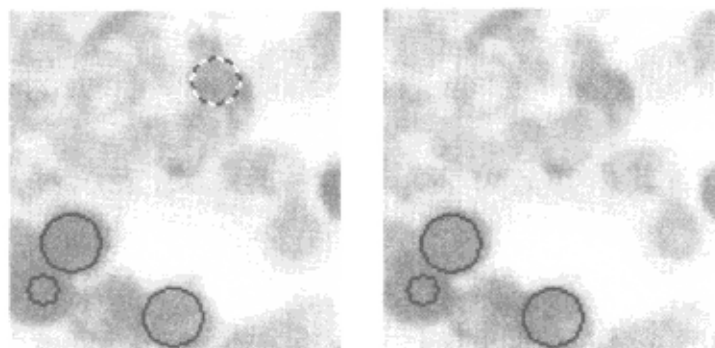


图 9-7 噪声引起的伪细胞

2) 去除同两个或两个以上的圆相交并且其相交部分的面积占总面积的很大比例的圆, 即自己本身不跟别的圆相交的部分占总面积的比例太小, 可能是细化不精确所导致的 (如图 9-8 所示)。

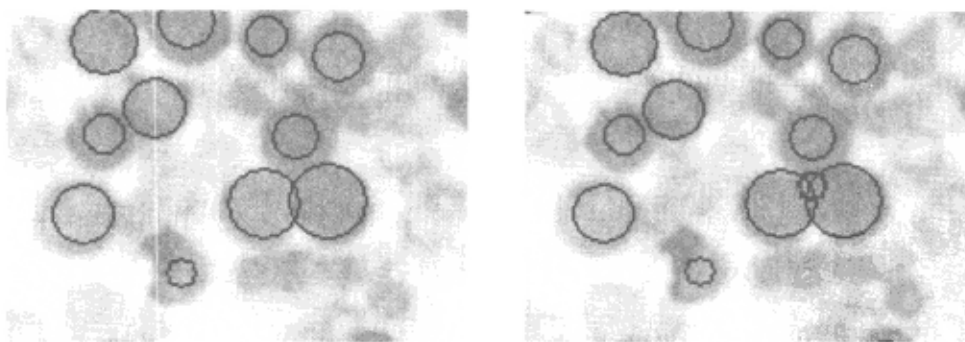



图 9-8 深色的两个像素宽的圆标志

3) 去除被包含的圆, 有时候一个大圆内包含一个较小的圆, 相对应的在图像中是一个大细胞覆盖了一个小细胞, 这种情况连肉眼也辨认不出来, 计算机更是不可能, 所以应该去掉一个圆, 这样计算机计数才能符合实际。

4) 如果某个圆内有的像素偏离了阈值范围, 一般认为是存在误差。假设  $toobad$  表示远离阈值范围的像素数量,  $area$  为圆面积, 如果  $total > r_0 * r_0$  且  $toobad < total/6$ , 则为误差较大。如果  $toobad > total/6$ , 则认为失误过大, 应该去除。

最后, 经过形态学处理和修正, 红细胞相关的特征信息就会记录在相应的变量中, 显示特征值即可。

 腐蚀、膨胀次数是人为控制的, 需要细心选择以得到最好的结果。

### 9.1.5 边缘提取技术

所谓边缘就是图像中灰度发生急剧变化的像素点的集合。对二维图像进行边缘提取, 就是检测边缘在强度上的非连续性, 或者说对这些图像强度的不连续点进行提取。在图像中像素点的梯度可以很好地体现图像强度。

梯度的定义如下所示。

给定一幅二维图像  $f(x, y)$  在点  $(x, y)$  的梯度用幅度和方向表示为:

$$g(x, y) = \sqrt{G_x^2 + G_y^2} = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \quad (9-15)$$

其中  $g(x, y)$  表示梯度幅度,  $G_x$  表示对  $x$  的偏导,  $G_y$  表示对  $y$  的偏导。

$$\phi(x, y) = \arctan(G_y / G_x) \quad (9-16)$$

其中  $\phi(x, y)$  表示梯度方向。

梯度的方向就是函数的最大变化率的方向, 故梯度的数值就是  $f(x, y)$  在其最大变化率方向上的单位距离所增加的量。

对于二维图像, 偏导数可以用差分来近似, 如式 (9-17) 所示。

$$\begin{cases} \frac{\partial f(x, y)}{\partial x} \approx f(x+1, y) - f(x, y) \\ \frac{\partial f(x, y)}{\partial y} \approx f(x, y+1) - f(x, y) \end{cases} \quad (9-17)$$

进行上述近似后, 梯度的模如式 (9-18) 所示。

$$g(x, y) = \left\{ [f(x, y+1) - f(x, y)]^2 + [f(x+1, y) - f(x, y)]^2 \right\}^{1/2} \quad (9-18)$$

一般为了计算方便, 采用式 (9-19) 绝对值近似公式:

$$g(x, y) \approx |f(x+1, y) - f(x, y)| + |f(x, y+1) - f(x, y)| \quad (9-19)$$

对于彩色图像, 可以分别对  $R$ 、 $G$ 、 $B$  进行梯度计算, 用  $g(x, y) = g_R + g_G + g_B$  作为梯度值。

在离散情况下多用梯度算子来检测提取边缘。常用的梯度算子如表 9-1 所示。

在系统操作过程中, 图像中往往存在这样的点, 覆盖在多个细胞的边界相连处, 使得阈值分割后将边界之间的缝隙也选为目标物体, 这样就使得几个细胞成为一个整体无法区分。因为边界相连处和细胞内部的颜色值差距较大存在突变, 而存在颜色突变的地方其相应的梯度值也较大, 因此, 梯度修正就是去掉这些梯度值很大的点, 使其不被标志。在修正中采用 Sobel 算子。图 9-9a

为阈值分割后的图像，图 9-9b 为梯度修正后的效果。

表 9-1 常用算子比较

算子名	模 板	特 性
罗伯特 (Roberts) 算子	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$	对陡峭的低噪声图像响应最好，对噪声敏感
普鲁维特 (Prewitt) 算子	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$	微分对噪声有抑制作用
索贝尔 (Sobel) 算子	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	对灰度渐变和噪声较多的图像处理得较好
各项同性 Sobel 算子	$\begin{bmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$	检测沿不同方向边缘时梯度幅度一致

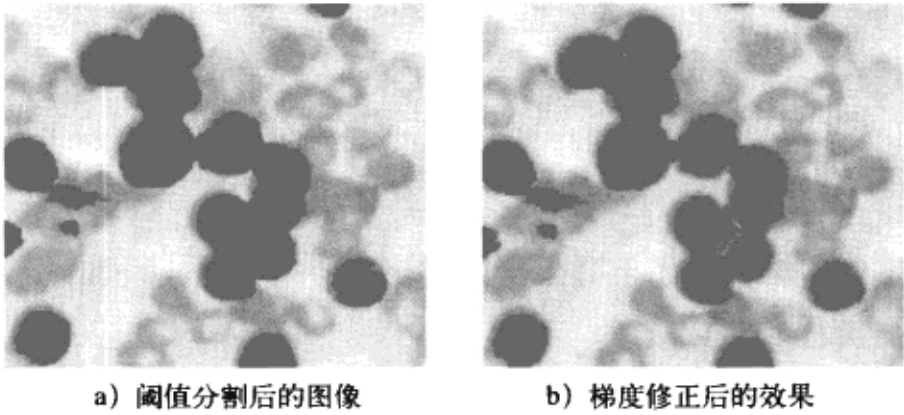


图 9-9 梯度修正前后对比图

## 9.2 系统功能

细胞检测与计数系统的主要功能是对血液红细胞进行检测与计数。经过图像平滑、阈值选取、梯度修正和查找中心点等一系列操作，识别出血液中的红细胞，同时统计红细胞的个数、平均半径和平均面积。

### 9.2.1 功能描述

细胞检测与计数系统主要实现以下功能：

- 1) 支持 BMP 格式的血液显微图像，能够保存处理后的图像。
- 2) 根据 HSI 阈值进行分割时，可以调整  $H$ 、 $S$ 、 $I$  三个分量误差。
- 3) 可重新载入原图像。

- 4) 能对图像进行平滑去噪。
- 5) 能对阈值分割后的目标图像进行颜色变换，默认为蓝色。
- 6) 能对选取后的图像进行孔洞填充。
- 7) 能进行梯度修正，而且可以显示 Sobel 边缘检测效果。
- 8) 能对个别未选入细胞进行人工选取或者对错选为细胞的区域进行人工去除。
- 9) 能对图像进行腐蚀或膨胀。
- 10) 能查找中心点，记录特征信息。
- 11) 能统计细胞个数、平均半径和平均面积等信息。
- 12) 可以选择显示方式：区域和边缘。

9.2.2 界面效果

细胞检测与计数系统的界面效果如图 9-10 所示，图中显示的是红细胞分割与计数结果。

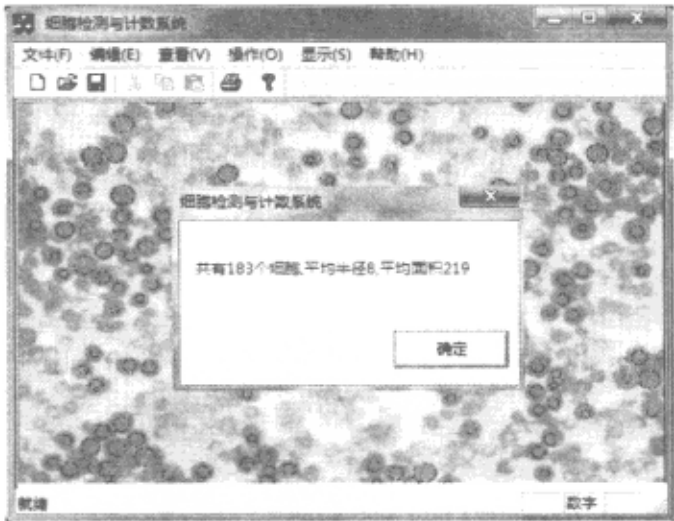


图 9-10 细胞检测与计数系统界面效果

9.3 系统结构与流程

细胞检测与计数系统主要分为图像效果增强模块、特征提取模块和显示方式模块。本节将介绍系统总体结构和主要流程。

9.3.1 总体结构

细胞检测与计数的总体结构如图 9-11 所示。

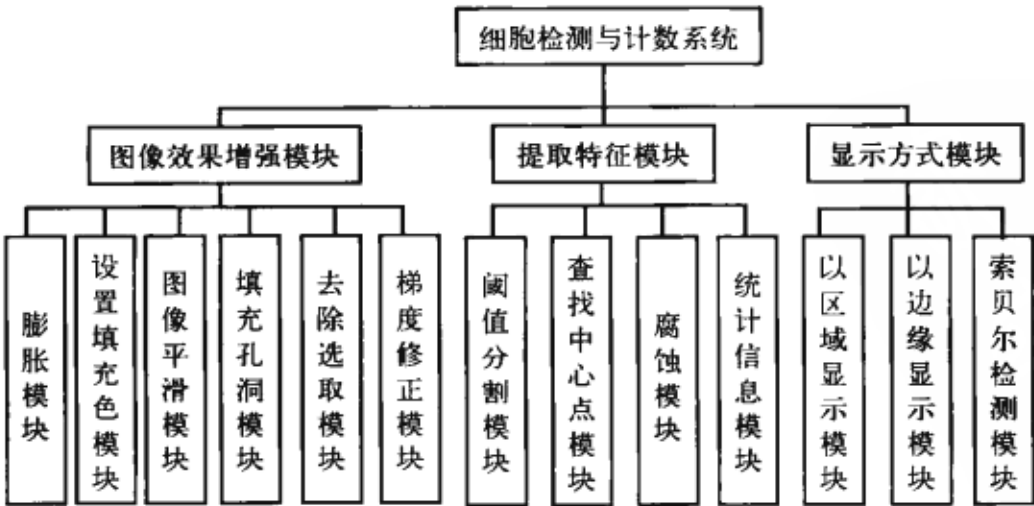


图 9-11 细胞检测与计数系统总体结构



### 9.3.2 主要流程

细胞检测与计数系统的主要流程如图 9-12 所示。

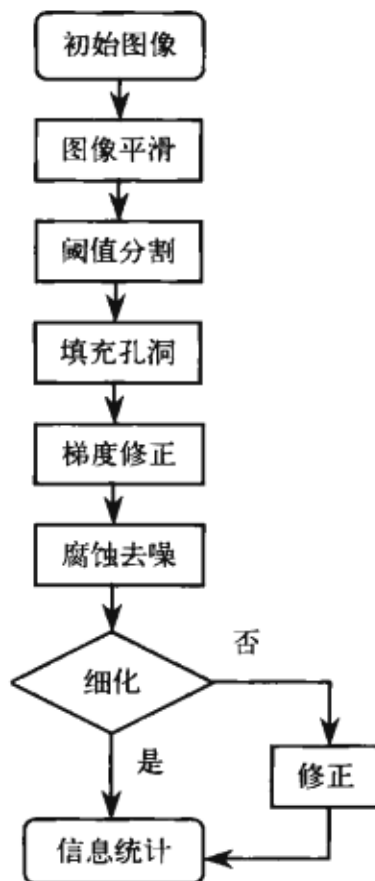


图 9-12 细胞检测与计数流程图

## 9.4 编程实现

细胞检测与技术系统采用 VC 2008 开发平台编程实现。

### 9.4.1 图像平滑模块

图像平滑模块主要调用 CCellView::OnProcSmooth()函数，对除边界外在图像中的每个像素点提取 RGB 的三个颜色通道值，在 8 邻域中进行求平均。

图像平滑的实现代码如下：

```

////////////////////////////////////
//进行图像平滑
////////////////////////////////////
void CCellView::OnProcSmooth()
{
    int wd,ht;

```

```

if(g_hBitmap)
{
    RGB *g_pTemp=new RGB[g_nMapWidth*g_nMapHeight];
    RGB *ptemp=g_pTemp;
    memset(g_pTemp,0,g_nMapWidth*g_nMapHeight*sizeof(RGB));
    RGB *cur=g_pImage;
    for (ht=0;ht<g_nMapHeight;ht++)
        for (wd=0;wd<g_nMapWidth;wd++)
        {
            //不包括边界点
            if (ht==0 || wd==0 || ht==g_nMapHeight-1 || wd==g_nMapWidth-1)
            { }
            else
            {
                //对红色分量的提取
                ptemp->r=( (cur-g_nMapWidth-1)->r +
                           (cur-g_nMapWidth )->r +
                           (cur-g_nMapWidth+1)->r +
                           (cur-1)->r +
                           cur->r +
                           (cur+1)->r +
                           (cur+g_nMapWidth-1)->r +
                           (cur+g_nMapWidth )->r +
                           (cur+g_nMapWidth+1)->r )/9;
                //对绿色分量的抽取
                ptemp->g=( (cur-g_nMapWidth-1)->g +
                           (cur-g_nMapWidth )->g +
                           (cur-g_nMapWidth+1)->g +
                           (cur-1)->g +
                           cur->g +
                           (cur+1)->g +
                           (cur+g_nMapWidth-1)->g +
                           (cur+g_nMapWidth )->g +
                           (cur+g_nMapWidth+1)->g )/9;
                //对蓝色分量的抽取
                ptemp->b=( (cur-g_nMapWidth-1)->b +
                           (cur-g_nMapWidth )->b +
                           (cur-g_nMapWidth+1)->b +
                           (cur-1)->b +
                           cur->b +
                           (cur+1)->b +
                           (cur+g_nMapWidth-1)->b +
                           (cur+g_nMapWidth )->b +
                           (cur+g_nMapWidth+1)->b )/9;
            }
            cur++;
            ptemp++;
        }
    }

```

```

    }
    memcpy(g_pImage,g_pTemp,g_nMapWidth*g_nMapHeight*sizeof(RGB));
    delete[] g_pTemp;
    InvalidateRect(0,TRUE);
}
else
    MessageBox("请先打开图像文件!");
}

```

### 9.4.2 HSI 阈值选取模块

首先选取一块红细胞区域,通过 DeHsiData()函数将 RGB 颜色空间转为 HSI 颜色空间并记录。然后得到此区域的  $H$ 、 $I$  分量的最大值和最小值,在整个图像范围内搜索满足条件的区域,即初始的红细胞区域。

HSI 阈值选取的实现代码如下:

```

////////////////////////////////////
void CCellView::ProcHSI(bool bEx)
{
    c_vHSI.clear();

    HSI *hsi;

    Huestep=256.0*g_nHuestep/100.0;
    if (Huestep>5.12)
        Huestep=5.12;
    Huestep+=0.0001;
    Intstep=g_nIntstep/100.0;
    if (Intstep>0.02)
        Intstep=0.02;
    Intstep+=0.0001;
    Satstep=g_nSatstep/100.0;
    if (Satstep>0.02)
        Satstep=0.02;
    Satstep+=0.0001;

    c_SelectedRect.left+=scroll_lefttop.x;
    c_SelectedRect.right+=scroll_lefttop.x;
    c_SelectedRect.top+=scroll_lefttop.y;
    c_SelectedRect.bottom+=scroll_lefttop.y;
    for(int j = c_SelectedRect.top; j < c_SelectedRect.bottom; j++)
    {
        for(int i = c_SelectedRect.left; i < c_SelectedRect.right; i++)
        {
            if (i<0 || i>=g_nMapWidth || j<0 || j>=g_nMapHeight)
                continue;

```

```

        if(!g_pFlag[INDEX(i, j)].marked)
        {
            hsi=&g_pHSI[INDEX(i, j)];
            if(FindInVectorHSI(c_vHSI, *hsi) == false)
            {
                c_vHSI.push_back(*hsi);
                c_vAllSelected.push_back(*hsi);
            }
        }
    }

    if(c_vHSI.size())
    {
        memcpy(g_pFlagBack, g_pFlag,
            sizeof(FLAGS)*g_nMapWidth*g_nMapHeight);

        HsiProcess(bEx);
        GenEdge8();
        InvalidateRect(0, TRUE);
    }
}

////////////////////////////////////
void CCellView::DeHsiData()
{
    int      wd, ht;
    HSI      tmp;
    RGB      *cur;

    if(g_pHSI)
        delete[] g_pHSI;
    g_pHSI = new HSI[g_nMapWidth * g_nMapHeight];
    memset(g_pHSI, 0, sizeof(HSI) * g_nMapWidth * g_nMapHeight);

    HSI *pHueBuffer = g_pHSI;
    cur = g_pImage;
    for(ht = 0; ht < g_nMapHeight; ht++)
    {
        for(wd = 0; wd < g_nMapWidth; wd++)
        {
            RgbToHsi(cur, &tmp);
            pHueBuffer->Hue = tmp.Hue*255.0/360.0;
            pHueBuffer->Intensity = tmp.Intensity;
            pHueBuffer->Saturation = tmp.Saturation;
            pHueBuffer++;
            cur++;
        }
    }
}

```

```

    }
}
/////////////////////////////////////////////////////////////////
bool CCellView::FindInVectorHSI(vector<HSI> v, HSI c)
{
    int size = v.size();
    for(int i = 0; i < size; i++)
        if( fabs(c.Hue - v.at(i).Hue) < Huestep &&
            fabs(c.Intensity - v.at(i).Intensity) < Intstep &&
            fabs(c.Saturation - v.at(i).Saturation) < Satstep )
            return true;
    return false;
}
/////////////////////////////////////////////////////////////////
void CCellView::HsiProcess(bool bEx)
{
    RGB      *cur=g_pImage;
    HSI      *pHSI=g_pHSI;

    int      size = c_vHSI.size();
    double   sH, sS, sI, dH, dS, dI;
    HSI      tmp, vtmp;
    int      ht, wd;
    FLAGS    *cur_flag = g_pFlag;
    for(ht = 0; ht < g_nMapHeight; ht++)
    {
        for(wd = 0; wd < g_nMapWidth; wd++)
        {
            if(cur_flag->marked)
            {
                cur++;
                cur_flag++;
                pHSI++;
                continue;
            }
            memcpy(&tmp,pHSI,sizeof(HSI));
            pHSI++;
            for(int i = 0; i < size; i++)
            {
                vtmp = c_vHSI.at(i);
                sH = vtmp.Hue;
                sS = vtmp.Saturation;
                sI = vtmp.Intensity;
                dH = tmp.Hue;
                dS = tmp.Saturation;
                dI = tmp.Intensity;
                if(!bEx)          // 排除法

```

```

        {
            if
            (
                sH - dH >= 0 - g_nHuestep * 2.55
                && sH - dH <= g_nHuestep * 2.55
                && sS - dS >= 0 - g_nSatstep / 100.0
                && sS - dS <= g_nSatstep / 100.0
                && sI - dI >= 0 - g_nIntstep / 100.0
                && sI - dI <= g_nIntstep / 100.0
            )
            {
                cur_flag->marked = 1;    // 设置标志
                continue;
            }
        }
    else
    {
        if
        (
            (sH - dH <= 0 - g_nHuestep * 2.55
              || sH - dH >= g_nHuestep * 2.55)
            && (sS - dS <= 0 - g_nSatstep / 100.0
              || sS - dS >= g_nSatstep / 100.0)
            && (sI - dI <= 0 - g_nIntstep / 100.0
              || sI - dI >= g_nIntstep / 100.0)
        )
        {
            cur_flag->marked = 1;    // 设置标志
            cur->r=cur->g=cur->b=0;
            continue;
        }
    }
    cur++;
    cur_flag++;
}

}

}

//////////////////////////////////////////////////
//RGB 彩色空间到 HSI 空间的转换
//////////////////////////////////////////////////
int RgbToHsi( RGB *Rgbhsi, HSI *Hsirgb)
{
    double R, G, B, Sum, Quo;
    double Radians, Angle, MinValue, MaxValue, TempD1, TempD2;
    R = ((double) Rgbhsi->r) / 255.0;
    G = ((double) Rgbhsi->g) / 255.0;

```



```

B = ((double) Rgbhsi->b) / 255.0;
Sum = R + G + B;
Hsirgb->Intensity = Sum / 3.0;
MinValue = (R < G) ? R : G;
MinValue = (B < MinValue) ? B : MinValue;
MaxValue = (R > G) ? R : G;
MaxValue = (B > MaxValue) ? B : MaxValue;
if(Hsirgb->Intensity < 0.00001)
    Hsirgb->Saturation = ZERO_SATURATION;
else
    Hsirgb->Saturation = 1.0 - (3.0 * MinValue) / Sum;
if(MinValue == MaxValue)
{
    Hsirgb->Hue = UNDEFINED_HUE;
    Hsirgb->Saturation = ZERO_SATURATION;
    return 0;
}
TempD1 = (((R - G) + (R - B)) / 2.0);
TempD2 = (R - G) * (R - G) + (R - B) * (G - B);
Quo = (TempD1 / sqrt(TempD2));
if(Quo > 0.9999999999999999)
    Radians = 0.0;
else if(Quo < -0.9999999999999999)
    Radians = 3.1415926535;
else
    Radians = acos(TempD1 / sqrt(TempD2));
Angle = Radians * DEGREES_PER_RADIAN;
if(B > G)
    Hsirgb->Hue = 360.0 - Angle;
else
    Hsirgb->Hue = Angle;
return 0;
}

```

### 9.4.3 梯度修正模块

梯度修正模块主要就是去除超出 Sobel 结果范围、亮度和色调的像素点，即将不符合目标物体特征的点设置其 *cur\_flag->marked=0*。从而部分细胞边界与边界之间的小空隙就会被去除，阈值分割后连为整体的细胞便可以区分开。

梯度修正的代码如下：

```

////////////////////////////////////
// 进行梯度修正
////////////////////////////////////
void CCellView::OnProcSobelCorrect()
{

```

```

if (!g_hBitmap)
{
    MessageBox("请先打开图片");
    return;
}
// backup
memcpy(g_pFlagBack, g_pFlag,
        sizeof(FLAGS)*g_nMapWidth*g_nMapHeight);

double max_intensity=0.0;
double min_hue=255.0;
double max_hue=0.0;
if (c_vAllSelected.size())
    for (int i=0; i<c_vAllSelected.size(); i++)
    {
        if (c_vAllSelected.at(i).Intensity>max_intensity)
            max_intensity=c_vAllSelected.at(i).Intensity;
        if (c_vAllSelected.at(i).Hue>max_hue)
            max_hue=c_vAllSelected.at(i).Hue;
        if (c_vAllSelected.at(i).Hue<min_hue)
            min_hue=c_vAllSelected.at(i).Hue;
    }
max_hue-=max_hue*0.1;
min_hue+=min_hue*0.1;

int x,y;
FLAGS *cur_flag=g_pFlag;
BYTE *cur_sobel=g_pSobel;
HSI *cur_hsi=g_pHSI;
for(y = 0; y < g_nMapHeight; y++)
    for(x = 0; x < g_nMapWidth; x++)
    {
        if (cur_flag->marked)
        {
            if (*cur_sobel > 40) // 参数需要调整
            {
                if (max_intensity>0.01)
                    // 亮度大于最大值并且色调超出范围
                    if (cur_hsi->Hue>max_hue || cur_hsi->Hue<min_hue)

                        cur_flag->marked=0;
            }
        }
        cur_hsi++;
        cur_flag++;
        cur_sobel++;
    }

```

```
GenEdge8();
InvalidateRect(0,TRUE);
```

#### 9.4.4 填充孔洞模块

填充孔洞模块包括三个部分,其中 void CCellView::FillHoles()为主程序,扫描整个图像,调用 ProcessFillHoles(int wd, int ht)函数对孔洞特征(如大小、是否标记、面积)进行记录,然后再调用 FilltheHoles(int wd, int ht) 函数根据上述孔洞特征进行填充。此模块算法相当于递归,由于次数过多,程序中引入了队列作为存储结构。

填充孔洞的代码如下:

```
void CCellView::FillHoles()
{
    int wd,ht;
    FLAGS *cur_flag=g_pFlag;

    // backup
    memcpy(g_pFlagBack,g_pFlag,
        sizeof(FLAGS)*g_nMapWidth*g_nMapHeight);

    qsz=g_nMapWidth*g_nMapHeight*2;
    qh=new int[qsz+1];
    if(!qh) return;

    for (ht=0;ht<g_nMapHeight;ht++)
        for (wd=0;wd<g_nMapWidth;wd++)
        {
            cur_flag->visited=0; // clear visited
            cur_flag++;
        }

    cur_flag=g_pFlag;
    for (ht=0;ht<g_nMapHeight;ht++)
        for (wd=0;wd<g_nMapWidth;wd++)
        {
            if (!cur_flag->marked && !cur_flag->visited)
            {
                g_nCellTotArea=0;
                ProcessFillHoles(wd,ht); //预处理孔洞
                if (g_nCellTotArea<MAX_HOLE && g_nCellTotArea>0)
                    FilltheHoles(wd,ht); //填充孔洞
            }
            cur_flag++;
        }
    delete[] qh;
    GenEdge8();
}
```

```

        InvalidateRect(0, TRUE);
    }

    //////////////////////////////////////
    //预处理孔洞
    //////////////////////////////////////
void CCellView::ProcessFillHoles(int wd, int ht)
{
    if (wd<0 || wd>g_nMapWidth-1 || ht<0 || ht>g_nMapHeight-1)
        return;
    qst=qh;
    memset(qst,0,qsz); //Clear the contents
    qs=qr=qst;
    *qs=xl=wd;
    qs++;
    *qs=yl=ht;
    qs++;
    g_pFlag[INDEX(xl,yl)].visited=1;
    g_nCellTotArea++;

    //Main queue loop
    while(qr!=qs)
    {
        if (yl>0)
        if(!g_pFlag[(yl-1)*g_nMapWidth+xl].visited &&
            !g_pFlag[(yl-1)*g_nMapWidth+xl].marked)
        {
            g_nCellTotArea++;
            *qs=xl;
            qs++;
            *qs=yl-1;
            qs++;
            g_pFlag[(yl-1)*g_nMapWidth+xl].visited=1;
        }
        //下
        if (yl<g_nMapHeight-1)
        if(!g_pFlag[(yl+1)*g_nMapWidth+xl].visited &&
            !g_pFlag[(yl+1)*g_nMapWidth+xl].marked)
        {
            g_nCellTotArea++;
            *qs=xl;
            qs++;
            *qs=yl+1;
            qs++;
            g_pFlag[(yl+1)*g_nMapWidth+xl].visited=1;
        }
        //左

```

```

        if (xl>0)
        if(!g_pFlag[yl*g_nMapWidth+xl-1].visited &&
            !g_pFlag[yl*g_nMapWidth+xl-1].marked)
        {
            g_nCellTotArea++;
            *qs=xl-1;
            qs++;
            *qs=yl;
            qs++;
            g_pFlag[yl*g_nMapWidth+xl-1].visited=1;
        }
        //右
        if (yl<g_nMapWidth-1)
        if(!g_pFlag[yl*g_nMapWidth+xl+1].visited &&
            !g_pFlag[yl*g_nMapWidth+xl+1].marked)
        {
            g_nCellTotArea++;
            *qs=xl+1;
            qs++;
            *qs=yl;
            qs++;
            g_pFlag[yl*g_nMapWidth+xl+1].visited=1;
        }

        //更新当前队列成员
        qr+=2;
        xl=*qr;
        yl=(qr+1);
    }
}

//////////////////////////////////////
//填充孔洞
//////////////////////////////////////
void CCellView::FilltheHoles(int wd, int ht)
{
    if (wd<0 || wd>g_nMapWidth-1 || ht<0 || ht>g_nMapHeight-1)
        return;
    g_nCellTotArea--;
    if (g_nCellTotArea<0)
        return;
    qst=qh;
    memset(qst,0,qsz); //Clear the contents
    qs=qr=qst;
    *qs=xl=wd;
    qs++;
    *qs=yl=ht;
    qs++;

```

```

g_pFlag[INDEX(x1,y1)].marked=1;

//Main queue loop
while(qr!=qs)
{
    if (y1>0)
    if(g_pFlag[(y1-1)*g_nMapWidth+x1].visited &&
        !g_pFlag[(y1-1)*g_nMapWidth+x1].marked)
    {
        g_nCellTotArea--;
        if (g_nCellTotArea<0)
            return;
        *qs=x1;
        qs++;
        *qs=y1-1;
        qs++;
        g_pFlag[(y1-1)*g_nMapWidth+x1].marked=1;
    }
    //下
    if (y1<g_nMapHeight-1)
    if(g_pFlag[(y1+1)*g_nMapWidth+x1].visited &&
        !g_pFlag[(y1+1)*g_nMapWidth+x1].marked)
    {
        g_nCellTotArea--;
        if (g_nCellTotArea<0)
            return;
        *qs=x1;
        qs++;
        *qs=y1+1;
        qs++;
        g_pFlag[(y1+1)*g_nMapWidth+x1].marked=1;
    }
    //左
    if (x1>0)
    if(g_pFlag[y1*g_nMapWidth+x1-1].visited &&
        !g_pFlag[y1*g_nMapWidth+x1-1].marked)
    {
        g_nCellTotArea--;
        if (g_nCellTotArea<0)
            return;
        *qs=x1-1;
        qs++;
        *qs=y1;
        qs++;
        g_pFlag[y1*g_nMapWidth+x1-1].marked=1;
    }
    //右

```



```

        if (yl<g_nMapWidth-1)
        if(g_pFlag[yl*g_nMapWidth+xl+1].visited &&
            !g_pFlag[yl*g_nMapWidth+xl+1].marked)
        {
            g_nCellTotArea--;
            if (g_nCellTotArea<0)
                return;
            *qs=xl+1;
            qs++;
            *qs=yl;
            qs++;
            g_pFlag[yl*g_nMapWidth+xl+1].marked=1;
        }
        //更新当前队列成员
        qr+=2;
        xl=*qr;
        yl=*(qr+1);
    }
}

```

#### 9.4.5 腐蚀模块

腐蚀模块中出现的八个方向即是前面核心技术中提到的方形模板，四个方向则是十字形模板。实现时循环使用这两个模板。

腐蚀操作的实现代码如下：

```

////////////////////////////////////
//腐蚀
////////////////////////////////////
void CCellView::OnProcErosion()
{
    if (g_bErosion)
        g_bErosion=false;
    else g_bErosion=true;

    // backup
    memcpy(g_pFlagBack,g_pFlag,
        sizeof(FLAGS)*g_nMapWidth*g_nMapHeight);

    // 腐蚀
    RGB *cur=g_pImage;
    FLAGS *cur_flag=g_pFlag;

    vector<long> addr;
    for (int ht=0;ht<g_nMapHeight;ht++)
        for (int wd=0;wd<g_nMapWidth;wd++)

```

```

{
    if (cur_flag->marked)
    {
        if (ht==0 || wd==0 || ht==g_nMapHeight-1 || wd==g_nMapWidth-1)
        {
            cur_flag->marked=0;
            cur++;
            cur_flag++;
            continue;
        }
        else if (g_bErosion)
        {
            if (
                !(cur_flag-1)->marked || // 左
                !(cur_flag+1)->marked || // 右
                !(cur_flag-g_nMapWidth-1)->marked || // 左上
                !(cur_flag-g_nMapWidth+1)->marked || // 右上
                !(cur_flag+g_nMapWidth-1)->marked || // 左下
                !(cur_flag+g_nMapWidth+1)->marked || // 右下
                !(cur_flag-g_nMapWidth)->marked || // 上
                !(cur_flag+g_nMapWidth)->marked ) // 下
                addr.push_back((long)cur_flag);
        }
        else
        {
            if (
                !(cur_flag-1)->marked || // 左
                !(cur_flag+1)->marked || // 右
                !(cur_flag-g_nMapWidth)->marked || // 上
                !(cur_flag+g_nMapWidth)->marked ) // 下
                addr.push_back((long)cur_flag);
        }
        cur++;
        cur_flag++;
    }

    int size=addr.size();
    for (int i=0;i<size;i++)
        ((FLAGS *)addr.at(i))->marked=0;
    addr.clear();
    GenEdge8();
    InvalidateRect(0,TRUE);
}

```

#### 9.4.6 边界生成模块

生成边界时，将像素点标记为边界的条件是点在图像中间且周围的点都没有被标记。

八方向生成边界的代码如下（四方向边界生成与之类似）：

```

////////////////////////////////////
// 八方向生成边界
////////////////////////////////////
void CCellView::GenEdge8()
{
    FLAGS    *cur = g_pFlag;
    int       width = g_nMapWidth;
    int       height = g_nMapHeight;
    for(int j = 0; j < g_nMapHeight; j++)
    {
        for(int i = 0; i < g_nMapWidth; i++)
        {
            cur->edged = 0;
            if(cur->marked)
            {
                if(j == 0 && i == 0) //左上
                {
                    if(!((cur + 1)->marked && (cur + width)->marked
                        && (cur + width + 1)->marked)) cur->edged = 1;
                }
                else if(j == 0 && i == width - 1) // 右上
                {
                    if(!((cur - 1)->marked && (cur + width)->marked
                        && (cur + width - 1)->marked)) cur->edged = 1;
                }
                else if(j == height - 1 && i == 0) // 左下
                {
                    if(!((cur + 1)->marked && (cur - width)->marked
                        && (cur - width + 1)->marked)) cur->edged = 1;
                }
                else if(j == height - 1 && i == width - 1) // 右下
                {
                    if(!((cur - 1)->marked && (cur - width)->marked
                        && (cur - width - 1)->marked)) cur->edged = 1;
                }
                else if(j == 0) // 上
                {
                    if
                    (
                        !((cur - 1)->marked
                        && (cur + 1)->marked
                        && (cur + width - 1)->marked
                        && (cur + width + 1)->marked
                        && (cur + width)->marked
                        )
                }
            }
        }
    }
}

```

```

        ) cur->edged = 1;
    }
    else if(i == 0) // 左
    {
        if
        (
            !((cur - width)->marked
            && (cur - width + 1)->marked
            && (cur + 1)->marked
            && (cur + width)->marked
            && (cur + width + 1)->marked
            )
        ) cur->edged = 1;
    }
    else if(j == height - 1) // 下
    {
        if
        (
            !((cur - 1)->marked
            && (cur + 1)->marked
            && (cur - width - 1)->marked
            && (cur - width + 1)->marked
            && (cur - width)->marked
            )
        ) cur->edged = 1;
    }
    else if(i == width - 1) // 右
    {
        if
        (
            !((cur - width)->marked
            && (cur - width - 1)->marked
            && (cur - 1)->marked
            && (cur + width)->marked
            && (cur + width - 1)->marked
            )
        ) cur->edged = 1;
    }
    else
    {
        if
        (
            !((cur - width - 1)->marked
            && (cur - width)->marked
            && (cur - width + 1)->marked
            && (cur - 1)->marked
            && (cur + 1)->marked
            )
        )
    }
}

```

```

        && (cur + width - 1)->marked
        && (cur + width)->marked
        && (cur + width + 1)->marked
    )
    ) cur->edged = 1;
}
}
cur++;
}
}
}

```

### 9.4.7 查找中心点和修正模块

查找中心点和修正模块分为两大部分，前一部分为“查找中心点”，通过腐蚀和细化得到临时中心点，判断这些中心点是噪声还是孤立点，若是噪声就去除，若是孤立点就记录下其特征值。还有一种情况就是中心点相近，这可能是由于操作误差造成的，需要进行平均化以得到真正的中心点。后一部分为“修正”，程序流程基本符合核心技术中提到的几个方面。

修正和查找细胞中心点的代码如下：

```

////////////////////////////////////
//修正和查找细胞中心点
////////////////////////////////////
void CCellView::OnProcFindCenter()
{
    if(g_hBitmap)
    {
        // backup
        memcpy(g_pFlagBack,g_pFlag,
            sizeof(FLAGS)*g_nMapWidth*g_nMapHeight);

        int pre_shrink_count;
        CPre setdlg;
        if (IDOK==setdlg.DoModal()) // 对话框用于输入腐蚀次数
        {
            pre_shrink_count=setdlg.c_nPreCount;
        }
        else return;

        int i,j,x,y;
        FLAGS *cur_flag;
        // 先去掉pre_shrink_count层皮
        GenEdge8();
        for (int i=0;i<pre_shrink_count;i++)
        {
            cur_flag=g_pFlag;

```



```

        for(y = 0; y < g_nMapHeight; y++)
            for(x = 0; x < g_nMapWidth; x++)
            {
                // 去掉边界
                if (cur_flag->edged)
                    cur_flag->marked=0;
                cur_flag++;
            }
        if (i==0)
            GenEdge4();
        else
            GenEdge8();
    }
    InvalidateRect(NULL, TRUE);
    MessageBox("去掉了噪声");

    cur_flag=g_pFlag; // 清除 visited 标志
    for(y = 0; y < g_nMapHeight; y++)
        for(x = 0; x < g_nMapWidth; x++)
        {
            cur_flag->visited=0;
            cur_flag->center=0;
            cur_flag++;
        }

    GenEdge8();
    CENTER_POINT pt;
    points_temp.clear();
    bool changed;

    for (i=0;i<40;i++) // 标志中心点的腐蚀
    {
        changed=false;
        // 清除 visited 标志
        cur_flag=g_pFlag;
        for(y = 0; y < g_nMapHeight; y++)
            for(x = 0; x < g_nMapWidth; x++)
            {
                cur_flag->visited=0;
                cur_flag++;
            }
        cur_flag=g_pFlag;
        for(y = 0; y < g_nMapHeight; y++)
            for(x = 0; x < g_nMapWidth; x++)
            {
                if (y>0 && y<g_nMapHeight-1 && x>0
                    && x<g_nMapWidth-1) // 最边上的不用处理

```



```

    {
        c_bEdge=true;
        // 没有访问过的边界
        if (cur_flag->edged && !cur_flag->visited)
        {
            if ( !(cur_flag-1)->marked &&
                !(cur_flag+1)->marked&&
                !(cur_flag+g_nMapWidth)->marked&&
                !(cur_flag-g_nMapWidth)->marked )
            {
                if (i==0) // 基本上是噪音
                {
                    cur_flag++;
                    continue;
                }
                // 孤立的点
                cur_flag->center=1;
                // 保存一下 CENTER_POINT 信息
                pt.x=x;
                pt.y=y;
                pt.radius=i+4+pre_shrink_count*2;
                points_temp.push_back(pt);

                cur_flag++;
                continue;
            }
            else
                MarkIt(x,y); // 判断是否需要保存

            if (c_bEdge) // 需要保存
                SaveIt(x,y,i+6);
        }
    }
    cur_flag++;
}

cur_flag=g_pFlag;
for(y = 0; y < g_nMapHeight; y++)
    for(x = 0; x < g_nMapWidth; x++)
    {
        // 去掉边界
        if (cur_flag->edged)
        {
            changed=true;

```

```

        cur_flag->marked=0;
    }
    cur_flag++;
}
if (i%2==0)
    GenEdge4();
else
    GenEdge8();

if (!changed)
    break;
}

// 清除 visited 标志
cur_flag=g_pFlag;
for(y = 0; y < g_nMapHeight; y++)
    for(x = 0; x < g_nMapWidth; x++)
    {
        cur_flag->visited=0;
        cur_flag++;
    }
// 取平均值, 获得中心点
vector<CENTER_POINT> points;
cur_flag=g_pFlag;
for(y = 0; y < g_nMapHeight; y++)
    for(x = 0; x < g_nMapWidth; x++)
    {
        // 最边上的不用处理
        if (y>0 && y<g_nMapHeight-1 && x>0 && x<g_nMapWidth-1)
        {
            if (cur_flag->center)
            {
                if ( !(cur_flag-1)->center && !(cur_flag+1)->center &&
                    !(cur_flag+g_nMapWidth)->center &&
                    !(cur_flag-g_nMapWidth)->center &&
                    !(cur_flag+g_nMapWidth-1)->center &&
                    !(cur_flag+g_nMapWidth+1)->center &&
                    !(cur_flag-g_nMapWidth-1)->center &&
                    !(cur_flag-g_nMapWidth+1)->center )
                {
                    // 孤立的点
                    pt.x=x;
                    pt.y=y;
                    points.push_back(pt);
                    cur_flag++;
                    continue;
                }
            }
        }
    }
}

```

```

        else
        {
            totarea=0;
            mradius=0;
            totx=0;
            toty=0;
            CalCellArea(x,y);
            pt.x=totx/totarea;
            pt.y=toty/totarea;
            pt.radius=mradius;
            g_pFlag[pt.y*g_nMapWidth+pt.x].center=1;
            points.push_back(pt);
        }
    }
    cur_flag++;
}

m_vCenterPoints.clear();
int x0,y0;
bool adj;
// 清除 center 标志
cur_flag=g_pFlag;
for(y = 0; y < g_nMapHeight; y++)
    for(x = 0; x < g_nMapWidth; x++)
    {
        cur_flag->center=0;
        cur_flag++;
    }
// 平均化相近的中心点
for (i=0;i< points.size();i++)
{
    x0=points.at(i).x;
    y0=points.at(i).y;
    adj=false;
    for (j=i+1;j< points.size();j++)
    {
        x=points.at(j).x;
        y=points.at(j).y;
        if (abs(x0-x)+abs(y0-y)<10) // 相近
        {
            points.at(j).x=(x+x0)/2;
            points.at(j).y=(y+y0)/2;
            points.at(j).radius=points.at(i).radius;
            points.erase(points.begin()+j);
            i--;
            adj=true;
        }
    }
}

```

```

        break;
    }
}
if (!adj) // 非相近
{
    if (points.at(i).radius>5)
    {
        m_vCenterPoints.push_back(points.at(i));
        g_pFlag[points.at(i).y*g_nMapWidth+points.at(i).x].center=1;
    }
}

double max_intensity=0.0;
double hue, inte;
double min_hue=255.0;
double max_hue=0.0;
double overmax;
double overmin;
if (c_vAllSelected.size()>0) // 保存着的选项
{
    for (int i=0;i<c_vAllSelected.size();i++)
    {
        if (c_vAllSelected.at(i).Intensity>max_intensity)
            max_intensity=c_vAllSelected.at(i).Intensity;
        if (c_vAllSelected.at(i).Hue>max_hue)
            max_hue=c_vAllSelected.at(i).Hue;
        if (c_vAllSelected.at(i).Hue<min_hue)
            min_hue=c_vAllSelected.at(i).Hue;
    }

    InvalidateRect(0, TRUE);
    MessageBox("准备修正");
}
else
{
    InvalidateRect(0, TRUE);
    return;
}
max_hue+=max_hue*0.02;
min_hue-=min_hue*0.02;
overmax=max_hue*1.08;
overmin=min_hue*0.92;

int r0,r;
int tx,ty;
int area,toobad;

```

```

// 去掉被包含的圆
for (i=0;i<m_vCenterPoints.size();i++)
{
    x0=m_vCenterPoints.at(i).x;
    y0=m_vCenterPoints.at(i).y;
    r0=m_vCenterPoints.at(i).radius;
    for (j=i+1;j<m_vCenterPoints.size();j++)
    {
        x=m_vCenterPoints.at(j).x;
        y=m_vCenterPoints.at(j).y;
        r=m_vCenterPoints.at(j).radius;
        if (DISTANCE(x0,y0,x,y)<abs(r0-r)+4) // 包含
        {
            CDC *pdc=GetDC();
            CENTER_POINT centerp;
            CPen pen;
            pen.CreatePen(PS_DOT, 1, RGB(255,0,0));
            pdc->SelectObject(pen);
            if (r0>r) // 去掉 r0
                centerp=m_vCenterPoints.at(i);
            else
                centerp=m_vCenterPoints.at(j);
            Arc(pdc->m_hDC,
                centerp.x-scroll_lefttop.x-centerp.radius,
                centerp.y-scroll_lefttop.y-centerp.radius,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y+centerp.radius,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y
            );
            DeleteObject(pen);
            if (r0>r) // 去掉 r0
            {
                m_vCenterPoints.erase(m_vCenterPoints.begin()+i);
                i--;
            }
            else
                m_vCenterPoints.erase(m_vCenterPoints.begin()+j);
        }
    }
}
vector<CENTER_POINT> tocheck;
int n,size,total;
bool isok;
// 去掉潜在的错误(同两个圆相交,并且不相交的部分是噪声)

```

```

for (i=0;i<m_vCenterPoints.size();i++)
{
    tocheck.clear();
    x0=m_vCenterPoints.at(i).x;
    y0=m_vCenterPoints.at(i).y;
    r0=m_vCenterPoints.at(i).radius;
    for (j=0;j<m_vCenterPoints.size();j++)
    {
        if (i==j)
            continue;
        x=m_vCenterPoints.at(j).x;
        y=m_vCenterPoints.at(j).y;
        r=m_vCenterPoints.at(j).radius;
        if (DISTANCE(x0,y0,x,y)<abs(r0+r)) // 相交
        {
            pt.x=x; pt.y=y; pt.radius=r;
            tocheck.push_back(pt);
        }
    }
    size=tocheck.size();
    if (size>1) // 同两个以上的圆相交
    {
        area=0;
        total=0;
        toobad=0;
        for (tx=x0-r0;tx<x0+r0;tx++)
            for (ty=y0-r0;ty<y0+r0;ty++)
            {
                if (DISTANCE(x0,y0,tx,ty)<r0) // 所有圆内部的点
                {
                    if (tx<0 || tx>g_nMapWidth-1 || ty<0
                        || ty>g_nMapHeight-1)
                        continue;
                    isok=true;
                    for (n=0;n<size;n++)
                    {
                        pt=tocheck.at(n);
                        if (DISTANCE(tx,ty,pt.x,pt.y)<pt.radius)
                        {
                            isok=false;
                            break;
                        }
                    }
                    if (isok) // 同所有的圆都不相交的部分
                    {
                        total++;
                        hue=g_pHSI[ty*g_nMapWidth+tx].Hue;

```



```

        inte=g_pHSI[ty*g_nMapWidth+tx].Intensity;
        if (inte>max_intensity||hue>max_hue || hue
            < min_hue)
            area++;
        if (hue > overmax || hue < overmin)
            toobad++;
    }
}
}
if (total<r0*r0 || (total<r0*r0*1.5 && area>total*0.5) || toobad>total/6)
//需要调整
{
    CDC *pdc=GetDC();
    CENTER_POINT centerp;
    CPen pen;
    if (total<r0*r0) // 红色表示面积过小
        pen.CreatePen(PS_SOLID, 2, RGB(255,0,0));
    else if (toobad>total/6) // 绿色表示错误点过多
        pen.CreatePen(PS_SOLID, 2, RGB(0,255,0));
    else // 蓝色表示误差过大
        pen.CreatePen(PS_SOLID, 2, RGB(0,0,255));
    pdc->SelectObject(pen);
    centerp=m_vCenterPoints.at(i);
    Arc(pdc->m_hDC,
        centerp.x-scroll_lefttop.x-centerp.radius,
        centerp.y-scroll_lefttop.y-centerp.radius,
        centerp.x-scroll_lefttop.x+centerp.radius,
        centerp.y-scroll_lefttop.y+centerp.radius,
        centerp.x-scroll_lefttop.x+centerp.radius,
        centerp.y-scroll_lefttop.y,
        centerp.x-scroll_lefttop.x+centerp.radius,
        centerp.y-scroll_lefttop.y
    );
    DeleteObject(pen);
    m_vCenterPoints.erase(m_vCenterPoints.begin()+i);
    i--;
}
}
}
for (i=0;i<m_vCenterPoints.size();i++)
{
    r0=m_vCenterPoints.at(i).radius;
    if (r0<10)
    {
        x0=m_vCenterPoints.at(i).x;
        y0=m_vCenterPoints.at(i).y;
        area=0;
    }
}

```

```

        toobad=0;
        for (tx=x0-r0;tx<x0+r0;tx++)
            for (ty=y0-r0;ty<y0+r0;ty++)
                if (sqrt(float ((x0-tx)*(x0-tx)+(y0-ty)*(y0-ty)))<r0)
                {
                    if(tx<0||tx>g_nMapWidth-1||ty<0||
                        ty>g_nMapHeight-1)
                        continue;
                    hue=g_pHSI[ty*g_nMapWidth+tx].Hue;
                    if (hue > max_hue || hue < min_hue)
                        area++;
                    if (hue > overmax || hue < overmin)
                        toobad++;
                }
        if (area>r0*r0 || toobad>r0*r0/2) //需要调整
        {
            CDC *pdc=GetDC();
            CENTER_POINT centerp;
            CPen pen;
            if (toobad>r0*r0/2)
                pen.CreatePen(PS_DOT, 1, RGB(0,128,0));
            else
                pen.CreatePen(PS_DOT, 1, RGB(0,0,255));
            pdc->SelectObject(pen);
            centerp=m_vCenterPoints.at(i);
            Arc(pdc->m_hDC,
                centerp.x-scroll_lefttop.x-centerp.radius,
                centerp.y-scroll_lefttop.y-centerp.radius,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y+centerp.radius,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y,
                centerp.x-scroll_lefttop.x+centerp.radius,
                centerp.y-scroll_lefttop.y
            );
            DeleteObject(pen);
            m_vCenterPoints.erase(m_vCenterPoints.begin()+i);
            i--;
        }
    }
}
// 去掉潜在的错误(相交,并且包含许多色调范围外的像素)
for (i=0;i<m_vCenterPoints.size();i++)
{
    x0=m_vCenterPoints.at(i).x;
    y0=m_vCenterPoints.at(i).y;
    r0=m_vCenterPoints.at(i).radius;

```

```

for (j=i+1;j<m_vCenterPoints.size();j++)
{
    x=m_vCenterPoints.at(j).x;
    y=m_vCenterPoints.at(j).y;
    r=m_vCenterPoints.at(j).radius;
    if (sqrt((float)(x0-x)*(x0-x)+(y0-y)*(y0-y))<abs(r0+r)) // 相交
    {
        area=0;
        toobad=0;
        for (tx=x0-r0;tx<x0+r0;tx++)
            for (ty=y0-r0;ty<y0+r0;ty++)
        if (sqrt((float)(x0-tx)*(x0-tx)+(y0-ty)*(y0-ty))<r0)
        {
            if(tx<0||tx>g_nMapWidth-1||ty<0||ty>g_nMapHeight-1)
                continue;
            hue=g_pHSI[ty*g_nMapWidth+tx].Hue;
            if (hue > max_hue || hue < min_hue)
                area++;
            if (hue > overmax || hue < overmin)
                toobad++;
        }
        if (area>r0*r0 || toobad>r0*r0/2) // need adjust
        {
            InvalidateRect(0,TRUE);
            {
                CDC *pdc=GetDC();
                CENTER_POINT centerp;
                CPen pen;
                if (toobad>r0*r0/6)
                    pen.CreatePen(PS_SOLID,3,
                                RGB(255,0,0));
                else
                    pen.CreatePen(PS_SOLID,3, RGB(0,0,0));
                pdc->SelectObject(pen);
                centerp=m_vCenterPoints.at(i);
                Arc(pdc->m_hDC,
                    centerp.x-scroll_lefttop.x-centerp.radius,
                    centerp.y-scroll_lefttop.y-centerp.radius,
                    centerp.x-scroll_lefttop.x+centerp.radius,
                    centerp.y-scroll_lefttop.y+centerp.radius,
                    centerp.x-scroll_lefttop.x+centerp.radius,
                    centerp.y-scroll_lefttop.y,
                    centerp.x-scroll_lefttop.x+centerp.radius,
                    centerp.y-scroll_lefttop.y
                    );
                DeleteObject(pen);
            }
        }
    }
}

```

```
        m_vCenterPoints.erase(m_vCenterPoints.begin()+i);  
        i--;  
        Sleep(2000);  
        break;  
    }  
    }  
    }  
    Sleep(10000);  
    InvalidateRect(0, TRUE);  
}  
else  
    MessageBox("请先打开图像文件!");  
}
```

## 9.5 经验分享

在细胞检测与技术系统中较为消耗时间的两个部分是“图像分割”和“查找中心点”。对于HSI 阈值选取，如果选择的颜色范围过大，会消耗较长的时间。因为对转换来的颜色空间，要用其  $H$ 、 $S$ 、 $I$  三个分量对整幅图像的所有像素点逐一进行阈值比较，运算量大、耗费时间长。因此在阈值分割时，设置了误差选择，可以根据误差大小去掉一些特征色，以减少运算量、节省运行时间。对于“查找中心点”操作，因为要考虑原来临时中心点是否为噪声、孤立点或临近点等，所以比较费时。这些都是程序需要优化的部分。

## 第 10 章 指纹提取与识别系统

“一斗穷，二斗富，三斗四斗卖豆腐……”不少中国孩子都是在咿呀学语的时候扳着十个指头念着这首童谣开始“研究”指纹的。考古证实，远在公元前 7000 年到公元前 6000 年的时候，指纹便已在中国开始使用，那时的陶艺匠人在制作陶器时会较有规律地按上自己的指纹形成一种装饰效果。史料记载，中国人从西周时期开始利用指纹来签文书、立契约。1896 年阿根廷成为世界上第一个以指纹为依据进行罪犯身份鉴别的国家。作为一种可靠的身份鉴别手段，指纹识别是生物特征识别技术中发展最早、应用最广的。本章围绕一个指纹提取与识别系统来介绍指纹自动识别的核心技术原理和编程实现方法。

**本章要点：**

- 指纹图像场及其分割技术
- 指纹图像增强技术
- 指纹图像细化技术
- 指纹局部特征点提取技术
- 指纹匹配技术
- 指纹提取与识别系统功能描述
- 指纹提取与识别系统的总体结构与主要流程
- 指纹提取与识别系统的编程实现

### 10.1 核心技术原理

指纹提取与识别系统涉及的核心技术有指纹图像场及其分割技术、指纹图像增强技术、指纹图像细化技术、指纹细节点提取技术和指纹匹配技术。

#### 10.1.1 指纹图像场及其分割技术

人的皮肤由表皮、真皮和皮下组织三部分组成，指纹就是表皮上突起的纹线。指纹之所以能用于身份鉴别，是因为它具备普遍性（人人皆有）、唯一性（各不相同）和稳定性（终生不变）

的特征。指纹识别要先从采集到的指纹图像中把纹线和背景分割开,然后提取出纹线的走向、明暗、疏密等特征,最后依据特征进行比对以判断两个指纹是否相同。指纹的这些特征可以用图像场来刻画。

### 1. 指纹图像场

指纹图像场是一个能呈周期性变化的特殊波动场,由 4 个场来描述,分别为强度场、梯度场、方向场和频率场。

#### (1) 强度场

在指纹图像场中强度场描述的是指纹纹线在某点的明暗程度,强度场越大表示指纹纹线在该点处的明亮程度越强。在采集指纹图像时,不同的传感器所获得的指纹强度场场强类型不同。如常用的光学图像其指纹强度场就可用灰度场来描述。

给定一幅二维指纹图像  $f(x, y)$ , 设其强度场为  $I$ 。则其灰度场强为:

$$|\vec{I}(x, y)| = f(x, y) \quad (10-1)$$

其中  $f(x, y)$  为指纹图像在该点的灰度值大小。

#### (2) 梯度场

在指纹图像场中,梯度形象上的描述是指纹纹线隆起的最陡的程度。对图像中某一点做偏微分,则该偏微分的方向即是最陡的方向也是梯度场的方向。设图像梯度场为

$T(x, y)$ , 则:

$$T(x, y) = \nabla f(x, y) = \left( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y} \right) \quad (10-2)$$

指纹图像梯度场大小为:

$$|T(x, y)| = \sqrt{G_x^2 + G_y^2} \quad (10-3)$$

其中  $G_x$  表示对  $x$  的偏导,  $G_y$  表示对  $y$  的偏导。

#### (3) 方向场

方向场的大小是梯度场正交分解参数  $\theta$  的值,其方向按右手定理垂直于梯度方向。理想的方向场大小为:

$$\sin \theta = \frac{|G_x(x, y)\vec{i}|}{|\vec{T}|} \quad (10-4)$$

$$\cos \theta = \frac{|G_y(x, y)\vec{j}|}{|\vec{T}|} \quad (10-5)$$

其中  $\vec{i}, \vec{j}$  为正交坐标的单位向量。从中可以推导出方向场大小为:

$$\theta(x, y) = \frac{1}{2} \arctan(\tan 2\theta) \quad (10-6)$$



在实际的指纹图像方向场中，任何一点的像素都必须与周围像素联系起来（如八邻域），才能说明该点在图像中的性质。因此，需要将周围像素点的灰度值累加再求平均以趋近真实。设：

$$O_x(x, y) = \sum_{s=i-\frac{n}{2}}^{i+\frac{n}{2}} \sum_{t=j-\frac{n}{2}}^{j+\frac{n}{2}} 2G_x(s, t)G_y(s, t) \quad (10-7)$$

$$O_y(x, y) = \sum_{s=i-\frac{n}{2}}^{i+\frac{n}{2}} \sum_{t=j-\frac{n}{2}}^{j+\frac{n}{2}} [G_x^2(s, t) - G_y^2(s, t)] \quad (10-8)$$

其方向场的大小为：

$$\theta(x, y) = \frac{1}{2} \arctan \frac{V_x(x, y)}{V_y(x, y)} \quad (10-9)$$

其中  $(x, y)$  的偏导数是通过 Sobel 算子模板来求得的。x 与 y 方向的模板分别为：

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}。由 Sobel 算子求得的指纹图像方向场角度会偏转  $135^\circ$ ，因此在$$

计算完之后要再减去  $135^\circ$ 。

#### (4) 频率场

设  $g$  为图像中某点  $(x, y)$  的频率，则  $g$  就为该点方向场垂直方向上单位长度内脊线的条数。上述定义是由 Miao 和 Maltoni (1997) 把指纹图像脊谷形成的横截面看成是正弦函数形状而得来的。则：

$$V(h) = \int_{x_1}^{x_2} |dh(x)/dx| dx \quad (10-10)$$

其中  $V(h)$  为  $[x_1, x_2]$  区间内灰度垂直变化的总和。 $h(x)$  为点  $(x, y)$  在方向场垂直方向上的灰度函数。

由此得到指纹图像的频率场为：

$$g = \frac{V(h)}{2(x_2 - x_1) \cdot a_m} = \frac{\int_{x_1}^{x_2} |dh(x)/dx| dx}{2(x_2 - x_1) \cdot a_m} \quad (10-11)$$

其中  $a_m$  为正弦波形的平均振幅。

给定一幅指纹图像如图 10-1 所示，指纹图像的梯度场和方向场分别如图 10-2 和图 10-3 所示。由梯度场可以看出整个指纹的轮廓，而由方向场则能判断出指纹图像中心点，该中心点与原图像中心点是吻合的，但是系统并不认为是指纹中心点，而是将其作为坐标系原点。

## 2. 指纹图像分割

指纹图像的分割指的是图像中指纹部分和背景部分的分割，其目的是除去背景冗余信息，大

大减少指纹预处理和特征提取时间，提高系统效率和精确度。下面通过强度场分割和梯度场分割来介绍一下指纹图像分割技术。



图 10-1 指纹原图像



图 10-2 指纹图像梯度场



图 10-3 指纹图像方向场

### (1) 基于强度场分割指纹图像

借助光学技术采集指纹是历史最久远并且使用最广泛的技术，其经过较长时间的应用考验，在一定程度上能够适应温度的变异，可达到 500DPI 的较高分辨率，且价格低廉，所以目前大多数指纹图像都是光学图像。光学图像强度场也可称为灰度场，所以光学指纹图像的强度场分布可以用图像的灰度直方图来表示。如图 10-4 所示为上图 10-1 指纹原图像的灰度直方图。

从图像场的角度看，强度场和梯度场反映纹线隆起及其变化，所以指纹本身的强度场值较高，其背景部分的强度场值则较低，指纹整体灰度值要高于背景部分。要选取的阈值对应于两部分的交界点，即直方图的谷底。在图 10-4 中，可以将灰度值在 $[0,100]$ 之间的看做背景部分的灰度值分布，而 $(100,255]$ 之间的为指纹灰度值分布。若选取阈值为 100，则图 10-1 指纹原图像强度场阈值分割效果如图 10-5 所示。

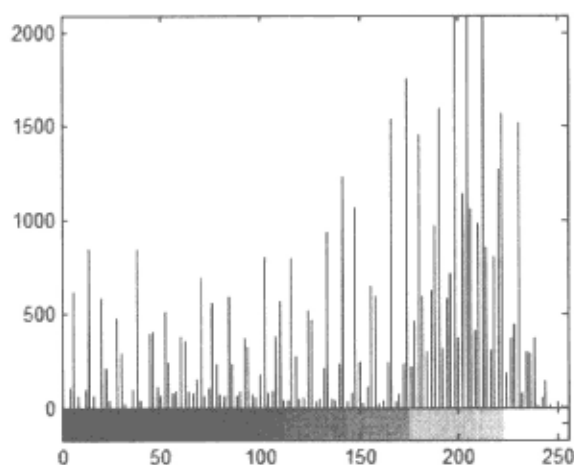


图 10-4 指纹原图像的灰度直方图



图 10-5 强度场阈值分割效果图

从图 10-5 可以看出，强度场阈值分割的效果并不十分理想。

### (2) 基于梯度场分割指纹图像

从图像场来看，指纹的梯度场值较高，其背景部分的梯度均值则较低，所以梯度场也可以用

来分割指纹图像。其实这种方法是分形维数分割指纹图像的演变。对图像而言,分形维数可以很好地表征纹理的粗糙程度和灰度分布的复杂程度,它也是单位区域上的梯度,通过指纹和背景之间分形维数的不同来区分两者。分形维数对尺度变化不敏感,这与人类视觉系统有很大相似之处。因此,分形维数常被用来描述图像的纹理特征而应用于图像分割、纹理识别和图像压缩等。

由于大多数分形并不是严格自相似的,因此需要寻求适合近似相似或者统计意义上相似的分形维数的定义方法。下面将介绍一种分形维数相似的方法——盒维法,它是目前实际应用中最广泛的一种方法。

盒维法其主要思想是将  $N \times N$  的图像分成  $m \times m$  的子块( $N/2 \geq m > 1, m \in Z$ ),令  $r = m/N$ ,将图像想象成三维空间中的曲面,  $xy$  表示平面,  $z$  表示灰度,  $xy$  平面被分割成许多  $m \times m$  的网格,在每个网格上是一列  $m \times m \times m$  的盒子。设图像灰度在第  $(i, j)$  网格中的最小值和最大值分别落在第  $k$  和第  $l$  个盒子中,  $n_r(i, j)$  是覆盖第  $(i, j)$  网格中的图像所需的盒子数,  $N_r$  是覆盖整个图像所需的盒子数,则:

$$n_r(i, j) = l - k + 1 \quad (10-12)$$

$$N_r = \sum_{i,j} n_r(i, j) \quad (10-13)$$

则分形维数的表达式为:

$$FW = \lim_{r \rightarrow 0} \frac{\lg N_r}{m} \quad (10-14)$$

要比较两个块的维数,由式(10-14)和式(10-13)推导可知,仅需要比较  $n_r(i, j)$  的大小,而  $n_r(i, j)$  的物理意义则是该块的梯度,所以,原先分形维数的比较转化为梯度的比较,即利用指纹和背景梯度场值不同来分割。

在应用中,通过“指纹图像场”一节中提到的方法求梯度场值后要进行梯度图像平滑,去除噪声,再选取合适的阈值进行分割。对于光学图像,阈值一般取 30~40。若按图像单位区域来计算梯度,则需要将单位区域上的平均梯度场值作为该区域阈值,然后再选取阈值进行分割。由于指纹和背景部分梯度场值不同,而指纹的梯度值整体偏高,因此基于梯度场值进行分割将得到很好的效果,如图 10-6 所示。



图 10-6 基于梯度场的图像分割

### 10.1.2 指纹图像增强技术

指纹图像增强技术是指对指纹图像采用一定的算法处理,使其纹理结构清晰化,尽量突出和保留固有的指纹特征信息,避免产生虚假特征,并能够消除噪声和减少计算量,以保证特征信息提取的准确性和可靠性。在指纹图像增强技术中,基于空间/频率域联合分析法主要包括 Gabor 变换法和小波变换法,两种方法的组合——Gabor 小波变换法,因同时考虑空域和频域两个因素而得到了广泛应用。

Gabor 小波变换法是以 Gabor 函数为母波函数得小波变换。基于 Gabor 函数的指纹图像增强, 依赖于纹线方向的准确估计。指纹图像在局部区域内几乎所有像素点的方向是一致的, 且在小波分解后小波低频系数的分布与原指纹图像的像素分布非常相似。因此, 低频系数很好地保留了原指纹图像的纹线方向和频率信息, 且抑制了高频扰动对纹线方向和频率信息的影响。由于 Gabor 函数具有频率选择和方向选择的特性, 采用 Gabor 函数不但能够去掉噪声, 还能把指纹的脊和谷的结构不失真地保留下来。常用的二维 Gabor 函数其数学模型为:

$$g(x, y) = \frac{1}{2\pi\sigma_x\sigma_y} \exp\left[-\frac{1}{2}\left(\frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}\right) + 2\pi if_0 x\right] \quad (10-15)$$

其中,  $f_0$  表示高斯函数的复调制频率,  $\sigma_x$  和  $\sigma_y$  分别是沿着  $x$  轴和  $y$  轴的高斯包络常量。

以 Gabor 函数  $g(x, y)$  为母函数, 对其进行适当的方向变换和尺度变换, 就得到 Gabor 小波滤波器。其数学模型为:

$$g_{uv}(x, y) = a^{-u} g(x', y') (a > 1, m, n \in Z) \quad (10-16)$$

其中,  $x' = a^{-u}(x \cos \theta + y \sin \theta)$ ,  $y' = a^{-u}(-x \sin \theta + y \cos \theta)$ ,  $\theta = v\pi/k$ ,  $k$  为总方向数,  $a^{-u}$  为尺度因子, 使得总能量与  $u$  无关,  $u$ 、 $v$  分别为尺度和方向。

因此, 只要调整  $u$ 、 $v$  的值就可以得到尺度和方向都不同的 Gabor 小波滤波器。

由于 Gabor 小波滤波器运算复杂度较高, 相对耗时。在本系统中, 采用简化后的 Gabor 小波滤波器, 纹线切线方向上进行平滑滤波, 模板为  $\frac{1}{7}(1, 1, 1, 1, 1, 1, 1)$ , 法线方向上进行锐化滤波, 模板为  $\frac{1}{7}(-3, -1, 3, 9, 3, -1, -3)$ 。通过不同方向上的图像增强, 使得谷线更白, 脊线更黑, 图像边缘更加平滑。效果如图 10-7 所示。

由图 10-7 可以看出, 整个指纹图像经过滤波在清晰度上得到了很大改进, 但是其灰度值范围仍然是  $[0, 255]$ , 即存在黑色 (灰度值 0) 和白色 (灰度值 255) 之间的其他灰度值, 因此表示脊和谷的灰度值有很多, 计算机很难区分出指纹脊和谷。这种情况下, 最常用的即是图像二值化方法。二值化的目的就是将一幅灰度图像转换为只有两种颜色的图像, 如黑白图像。

通常的二值化方法就是选定一个阈值, 大于阈值的设为白, 小于阈值的设为黑, 或者相反, 以得到黑白图像。针对本章, 根据指纹图像的特点: 指纹脊的部分沿纹线方向其邻近点都是黑点, 其垂直方向上邻近点都是白点; 而谷的部分其纹线方向邻近点都是白点, 其垂直方向上邻近点都是黑点。由此得出一个结论: 若纹线方向邻近像素点灰度值之和小于垂直方向上邻近像素点灰度值之和, 则为黑点 (脊上的点); 若大于则为白点 (谷上的点)。



图 10-7 指纹图像增强效果

设  $g(x_0, y_0)$  为某点的灰度值, 其方向场为  $O(x_0, y_0)$ ,  $a$  为一固定值。

切向像素值和为:

$$Hw = \sum_{x=x_0-a}^{x_0+a} f\left(x, \frac{x}{\cos(O(x_0, y_0))}\right) \quad (10-17)$$

法向像素值和为:

$$Vw = \sum_{x=x_0-a}^{x_0+a} f\left(x, \frac{x}{\sin(O(x_0, y_0))}\right) \quad (10-18)$$

若  $Hw < Vw$ , 则该点在脊线上。反之, 则在谷线上。

在系统中, 因为要考虑误差的存在, 采用加权求灰度和。纹线方向七点的权值为  $\{2, 2, 3, 4, 3, 2, 2\}$ 。纹线垂直方向七点的权值为  $\{1, 1, 1, 1, 1, 1, 1\}$ 。

经过二值化处理后的指纹图像效果如图 10-8 所示。

### 10.1.3 指纹图像细化技术

在指纹图像进行梯度场分割、图像增强和指纹图像二值化之后, 指纹图像成为高质量的黑白图像。但是其纹线宽度大多不止一个像素, 很多数据冗余, 增加了运算量。因此, 需要对指纹黑白图像进行细化处理。细化处理主要是针对指纹的脊部分, 在不改变图像像素拓扑关系的条件下, 循环去掉图像的边缘像素, 把纹线粗细不均匀的指纹图像转化成单像素线宽的中心线图像的过程。进行细化的主要目的是去除不必要的纹线信息, 使指纹图像的数据量及连接结构变得简单明了, 便于从指纹图像中提取细节特征, 从而提高指纹图像的匹配率。

细化算法有很多, 下面将介绍一种简单有效的算法, 它能够实现提取文本或者图像骨架的功能。从某像素点的八个邻域像素考虑, 看能否去掉这个点。图 10-9 给出了几个简单的例子。

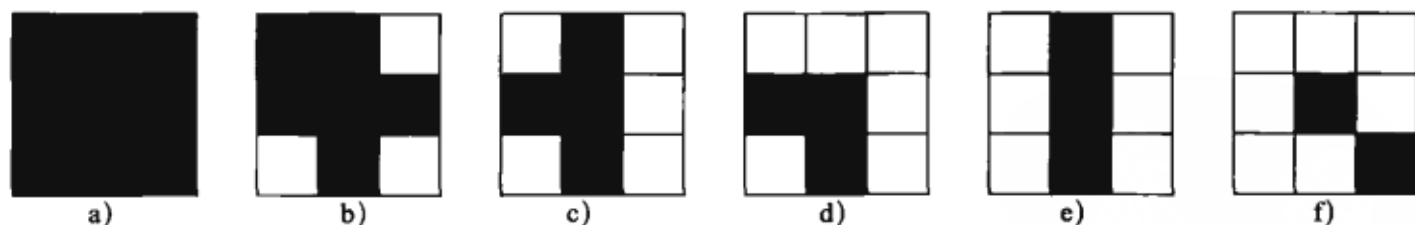


图 10-9 八邻域模型举例

$a$ 、 $b$  不能删, 因其为内部点, 去掉后中心出现孔洞, 改变属性。 $c$ 、 $d$  可以删, 去掉后仍是骨架。 $e$  不能删, 直线的骨架仍是直线, 去掉后改变了直线属性。 $f$  不能删, 因其为直线的端点, 如果删掉, 整个直线都会消失。

由此得出一个判断: 内部点、直线端点和孤立点都不可以删; 如果是边界点, 去掉后, 若连通分量个数不增加, 则此点可删除。



图 10-8 指纹图像二值化效果

可以用查表法来判断。如表 10-1 所示，表中从 0~255 总共 256 个索引，每一项都是 0 或 1。

表 10-1 细化判断表

uint8 erasetable[256]={			
	0,0,1,1,0,0,1,1,	1,1,0,1,1,1,0,1,	
	1,1,0,0,1,1,1,1,	0,0,0,0,0,0,0,1,	
	0,0,1,1,0,0,1,1,	1,1,0,1,1,1,0,1,	
	1,1,0,0,1,1,1,1,	0,0,0,0,0,0,0,1,	
	1,1,0,0,1,1,0,0,	0,0,0,0,0,0,0,0,	
	0,0,0,0,0,0,0,0,	0,0,0,0,0,0,0,0,	
	1,1,0,0,1,1,0,0,	1,1,0,1,1,1,0,1,	
	0,0,0,0,0,0,0,0,	0,0,0,0,0,0,0,0,	
	0,0,1,1,0,0,1,1,	1,1,0,1,1,1,0,1,	
	1,1,0,0,1,1,1,1,	0,0,0,0,0,0,0,1,	
	0,0,1,1,0,0,1,1,	1,1,0,1,1,1,0,1,	
	1,1,0,0,1,1,1,1,	0,0,0,0,0,0,0,0,	
	1,1,0,0,1,1,0,0,	0,0,0,0,0,0,0,0,	
	1,1,0,0,1,1,1,1,	0,0,0,0,0,0,0,0,	
	1,1,0,0,1,1,0,0,	1,1,0,1,1,1,0,0,	
	1,1,0,0,1,1,1,0,	1,1,0,0,1,0,0,0,	
};			

将要修改的某点其八邻域点的灰度值转为 0（代表黑色）或 1（代表白色）之后，按照下述规则转化为二进制八位数。最右下方的点为二进制数最高位第 8 位数，最下方中间位为第 7 位，左下方为第 6 位，该点右边的点为第 5 位，该点左边的点为第 4 位，右上方的点为第 3 位，最上边中间位为第 2 位，最左上方的点为二进制数最低位第 1 位数。这样得出二进制 8 位数之后，将其转化为十进制数，查出表中对应位置为 0 还是 1，若 0 则不修改，若 1 则将该中心点灰度值设为 255。将图 10-9 中列举的 6 种情况根据查表法得出结果如表 10-2 所示。

表 10-2 查表情况

标 号	二进制数	对应十进制数	查表后的数值	最终该点灰度值
<i>a</i>	00000000	0	0	0
<i>b</i>	10100100	164	0	0
<i>c</i>	10110101	181	1	255
<i>d</i>	10110111	183	1	255
<i>e</i>	10111101	189	0	0
<i>f</i>	01111111	127	0	0

通过表 10-2，就可以扫描整个图像，对每个非边界点按照规则进行计算，以判断该点是保留还是删除（灰度值设为 255）。如果有点被删除，则进行新一轮扫描，反复操作，直到没有点被删



除为止。通过查表法得到的细化效果如图 10-10 所示。

#### 10.1.4 指纹局部特征点提取技术

指纹特征主要分为总体特征和局部细节点特征。总体特征又有很多分类,如指纹形状特征、灰度分布特征及纹线频率特征等。统计结果表明,总体特征主要用来进行指纹分类和检索。而指纹特征提取中最常用的是局部细节点,指纹唯一性最终要靠细节特征来判别。还有一种很重要的指纹特征就是奇异点,它也是指纹图像匹配很好的参考点。下面将详细介绍局部细节点和奇异点这两种特征点的提取技术。



图 10-10 细化效果

##### 1. 局部细节点的提取

细节特征提取是指纹提取与识别系统中的重要环节,它既是对图像预处理效果的检验,又是后继指纹匹配算法实现的基础。指纹细节特征点主要是指纹线端点和纹线分叉点。纹线端点是纹线开始或结束的位置。纹线分叉点是指一条纹线分为两条或多条纹线的位置。

对细节点的提取主要采用八邻域法,用到的模板如图 10-11 所示。图中  $D$  为脊线上一个目标待检测点。通过式 (10-19) 可以推断  $D$  为端点还是分叉点。

$$T_D = \sum_{i=1}^7 |X_{i+1} - X_i| \quad (10-19)$$

其中  $X_i$  表示该像素点的灰度值。

$X_4$	$X_5$	$X_6$
$X_3$	$D$	$X_7$
$X_2$	$X_1$	$X_8$

图 10-11 八邻域模板

如果  $A$  点是端点,如图 10-12 所示,则将其放到  $D$  的位置上,去除  $A$  点后的八个邻域点里,必定还有一个黑点,按式 (10-20) 顺时针将每两个相邻像素点灰度值相减取绝对值,则应该有:

$$T_D = |X_5 - X_4| + |X_6 - X_5| = 2 \times 255 \quad (10-20)$$

所以通过八邻域法经过计算灰度值以判断  $T_D$  的大小若为  $2 \times 255$  则为端点。



图 10-12 端点模型及其细化后八邻域

同端点判断方法类似,分叉点的判断方法也是八邻域法,分叉点模型如图 10-13 所示。



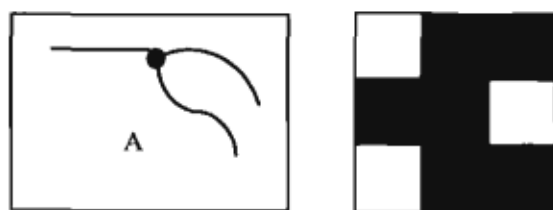


图 10-13 分叉点模型及其细化后八邻域

分叉点模型中除去中间要判断的点，四周八邻域点顺时针计算，由式 (10-21) 得：

$$T_D = |X_2 - X_1| + |X_3 - X_2| + |X_4 - X_3| + \dots = 6 \times 255 \quad (10-21)$$

由此，计算后当  $T_D$  大小为  $6 \times 255$  时中间的点为分叉点。

## 2. 奇异点的提取

奇异点通常分为中心点和三角点。中心点是指在指纹图的脊线流中最内层脊线上曲率最大的地方。三角点是指三个比较一致的脊线流交界的地方。奇异点位置相对稳定，指纹的位移和非线性形变对其影响不大。且奇异点存在具有普遍性，几乎所有的指纹图像中都存在奇异点。因此可以作为后续匹配算法的特征点。

奇异点检测方法有很多，Poincare index 方法是指纹奇异点检测中的常用方法之一。Poincare index 方法早在 1984 年被 Kawagoe 和 Togo 用来检测奇异点。本章的指纹提取与识别系统采用的是基于 Poincare index 和方向场相结合的方法，其离散模型为：

$$Poincare(x, y) = \frac{1}{2\pi} \sum_{k=0}^{N_p} \Delta(k) \quad (10-22)$$

其中：

$$\Delta(k) = \begin{cases} \delta(k), & \text{if } |\delta(k)| < \frac{\pi}{2}, \\ \pi + \delta(k), & \text{if } \delta(k) \leq -\frac{\pi}{2} \\ \pi - \delta(k), & \text{otherwise} \end{cases} \quad (10-23)$$

$$\begin{aligned} \delta(k) &= \Theta(\varphi_x(i'), \varphi_y(i')) - \Theta(\varphi_x(i), \varphi_y(i)), \\ i' &= (i+1) \bmod N_\varphi \end{aligned} \quad (10-24)$$

其中  $\Theta$  是方向场， $\varphi_x(i)$  和  $\varphi_y(i)$  表示点  $(x, y)$  附近曲线  $\varphi$  上第  $i$  个点的坐标。通过该区域的 Poincare 索引值，就能够确定该区域是否包含奇异点及其类型。在方向场中，若 Poincare 索引值为  $1/2$  则认定为中心点，为  $-1/2$  则为三角点。

方向域中的每一个点按公式 (10-22) 计算其相应的 Poincare 索引值，为减弱噪声对 Poincare 索引值计算的影响，分别采用  $5 \times 5$  和  $7 \times 7$  区域元素形成的闭合曲线，如图 10-14 所示，来计算

每个点的 Poincare 索引值, 只要其中一条的计算结果符合奇异点条件, 就认定该点为初始奇异点, 若两条曲线检测的奇异点类型不一致, 则直接认定该点为伪点。

用  $5 \times 5$  区域元素形成的闭合曲线来计算 Poincare 索引值, 有:

$$Poincare(i, j) = \sum_1^{12} |O_i - O_{(i+1) \bmod 12}| \quad (10-25)$$

$7 \times 7$  区域元素 Poincare 索引值计算与  $5 \times 5$  区域类似, 只是有 20 个点参与。因此, 只有在  $5 \times 5$  区域和  $7 \times 7$  区域形成的闭合曲线得到的 Poincare 索引值相同时, 求得的奇异点才是最终的奇异点。

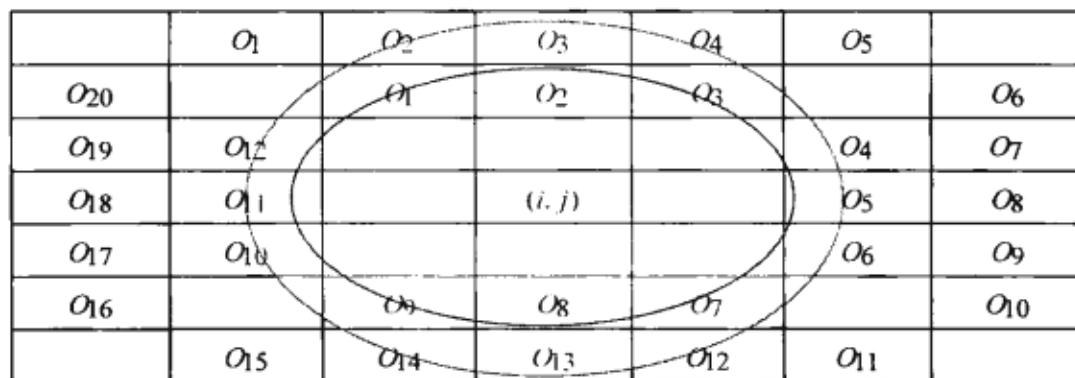


图 10-14  $5 \times 5$  区域和  $7 \times 7$  区域闭合曲线

标记出的纹线端点、分叉点和中心点效果如图 10-15 所示。

### 10.1.5 指纹匹配技术

提取特征点之后, 要判断两个指纹是否相同, 还要对两个指纹进行配准和匹配。

#### 1. 指纹配准

指纹的配准就是将对比的两个指纹图像进行对准叠合的过程。其目的就是使对比的两个指纹图像处于同一标准的坐标系下, 有相同的比对起始点, 这样后续的指纹匹配才有意义。把一个图像作为模板图像, 另一个进行比对的图像作为输入图像。通过指纹的拓扑结构, 将两个指纹图像的匹配转换为其特征点的匹配。

常用的指纹图像的配准方法有基于方向场的配准和特征点配准。

##### (1) 基于方向场的配准

对每个模板指纹的细节点  $P$  和每个输入指纹的细节点  $Q$ , 首先将其局部方向场进行匹配, 如果匹配分值大于预设的阈值, 再进一步对其局部细节点拓扑结构进行匹配, 并且根据两次匹配的结果决定它们是否可作为配准基准点。采用局部方向场预配准的优点在于指纹的方向场相对于指纹变形和噪声有较强的鲁棒性, 同源指纹之间相同区域的方向场很相似, 因此采用局部方向场可以较好地评价两指纹局部区域的相似性, 并能够确定出配准基点。但有些异源指纹也有很相似



图 10-15 指纹特征点标记

的方向场，因此采用局部方向场配准不够可靠。

## (2) 特征点配准


由于特征点在特征提取时已经存储，并记录了其相关特征信息，所以便于用来确定图像匹配的基准点。如果指纹图像中有中心点，且提取特征点无误，那么在指纹图像中就可以找到中心点作为坐标原点，然后再找到离中心点最近的特征点，由这两点确定极轴。若指纹图像中原本就没有中心点，则在图像中心附近查找最近的两个特征点，其中一个点作为原点，由这两点来确定极轴。由此可以算出两个指纹图像之间的平移量和旋转量，最终算得其他特征点相对于中心点的坐标和距离，将两个指纹图像进行配准。

## 2. 指纹匹配

在两个图像实现配准之后，便可以根据指纹特征点的拓扑结构来进行匹配。常用的方法有相似度计算法和界限盒模型等。

### (1) 相似度计算法

相似度计算法的公式是根据系统需要构造出来的，本章用到的相似度计算公式为： $4 \times \text{分数} \times \text{指纹特征最大相似数} / (\text{两个对比的指纹图像的特征点数和})^2$ 。其中分数是指模板指纹图像与输入指纹图像特征点匹配成功的表征，具体计算可以自行规定，最简单的就是有几对匹配成功就算这个为分数。

 相似度达到某个阈值就将两个指纹判断为相似。该阈值可通过多次实验得到一个经验值。

### (2) 界限盒法

由于两个指纹图像可能来自不同的采集设备，或者受图像预处理和噪声等的影响，原则上对应特征点完全匹配才算最好。但在操作中，匹配过程允许有一定的误差。下面就针对允许的误差部分介绍界限盒模型。

若模板图像中有某一个特征点  $A$ ，输入图像中有某一个特征点  $A'$ ，若完全匹配，则  $A = A'$ ，但存在误差，则称为相似匹配  $A \approx A'$ ，则  $A = A' + \Delta$ ， $\Delta$  即为界限盒模型代表的误差。

界限盒模型分为固定的和可变的，如图 10-16 所示。

其中  $angle\_size$  表示界限盒的角度变化范围， $radius\_size$  表示界限盒的半径变化范围。

使用固定大小的界限盒不能根据特征点周围邻域的情况进行调整，若一个特征点周围点较为密集则  $angle\_size$  和  $radius\_size$  都应该较小，而当周围邻域较为稀疏时，其两个表征值应该较大。

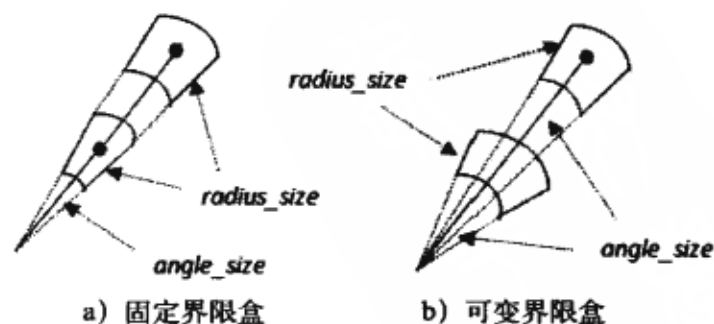


图 10-16 界限盒模型

所以鉴于固定大小的界限盒不能适应这种情况,现在常用的为可变大小的界限盒。可变大小的界限盒对非线性形变具有良好的鲁棒性。非线性形变一般在一个特定的区域内较大,然后非线性地向外扩张。当细节节点的极半径较小时,小的形变就可以造成大的极角的改变,而极半径的改变较小,所以界限盒的  $angle\_size$  应该比较大而  $radius\_size$  则应该比较小。另外,当细节节点的极半径较大时,极角的较小改变就会造成细节节点位置的较大变动,而极半径的形变可以看成是该细节节点与参照细节节点间的所有区域的形变的累加,在这种情况下,界限盒的  $angle\_size$  应该比较小而  $radius\_size$  则应该较大。

可变大小的界限盒其  $angle\_size$  和  $radius\_size$  计算公式如下:

$$angel\_size = \begin{cases} a\_small, a\_size < a\_small \\ a\_size, a\_small < a\_size < a\_large \\ a\_large, a\_size > a\_large \end{cases} \quad (10-26)$$

$$a\_size = \frac{\beta}{r^2} \quad (10-27)$$

$$radius\_size = \begin{cases} r\_small, r\_size < r\_small \\ r\_size, r\_small < r\_size < r\_large \\ r\_large, r\_size > r\_large \end{cases} \quad (10-28)$$

$$r\_size = r\_small + r/\alpha \quad (10-29)$$

其中  $a\_large$ 、 $a\_small$ 、 $r\_large$ 、 $r\_small$  分别为  $angle\_size$  和  $radius\_size$  的上下界,  $\alpha$ 、 $\beta$  是预先设定的。指纹特征点匹配流程如图 10-17 所示。

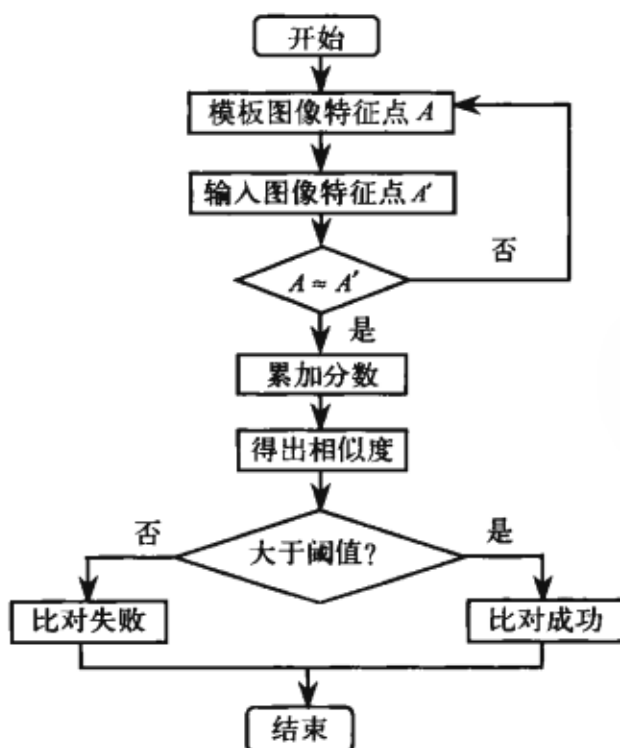


图 10-17 特征点匹配流程

## 10.2 系统功能

指纹提取与识别系统的主要功能是指纹匹配。通过指纹图像分割、增强和细化后提取指纹特征点，再通过指纹匹配识别两个指纹是否相同。

### 10.2.1 功能描述

指纹提取与识别系统主要实现以下功能：

- 1) 支持 BMP 格式的指纹图像，并能够保存处理结果。
- 2) 能对指纹图像进行梯度场和方向场变换。
- 3) 能对指纹图像进行平滑去噪。
- 4) 能利用梯度场进行指纹分割，区别出指纹本身和背景部分。
- 5) 能实现指纹图像灰度平衡。
- 6) 能通过缩小图像像素之间位置差异和灰度差异进行图像收敛。
- 7) 能对分割后的指纹图像实现效果增强。
- 8) 能对灰度指纹图像进行二值化，转化为高质量黑白图像。
- 9) 能对二值化后的指纹图像进行细化，转变为一个像素宽的黑白图像。
- 10) 能对细化后的指纹图像提取其局部细节点和奇异点，进行标记和存储。
- 11) 能去除伪指纹特征点。
- 12) 能进行两个指纹之间的匹配，输出判断结果。

### 10.2.2 界面效果

指纹提取与识别系统界面效果如 10-18 所示。



图 10-18 相同指纹情况下的效果图

## 10.3 系统结构与流程

指纹提取与识别系统由指纹图像预处理、指纹特征提取和指纹比对模块构成。

### 10.3.1 总体结构

指纹提取与识别系统的总体结构如图 10-19 所示。

### 10.3.2 主要流程

指纹提取与识别系统的流程图如图 10-20 所示。

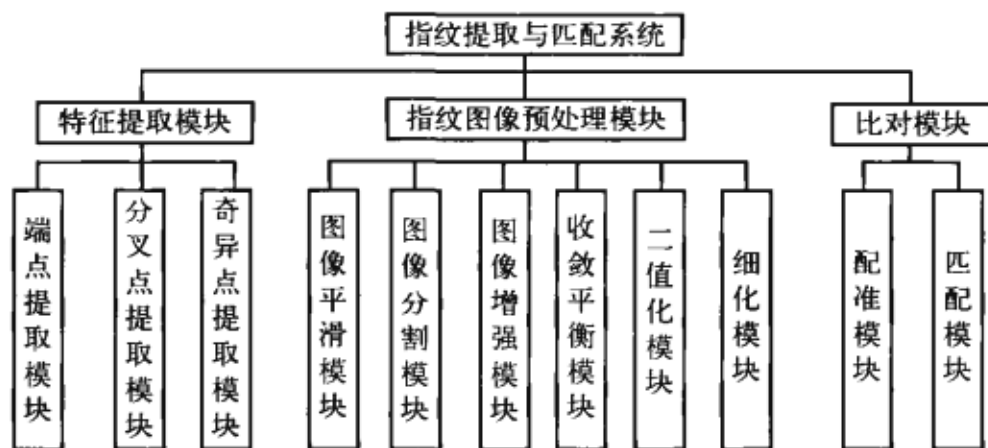


图 10-19 指纹提取与识别系统总体结构

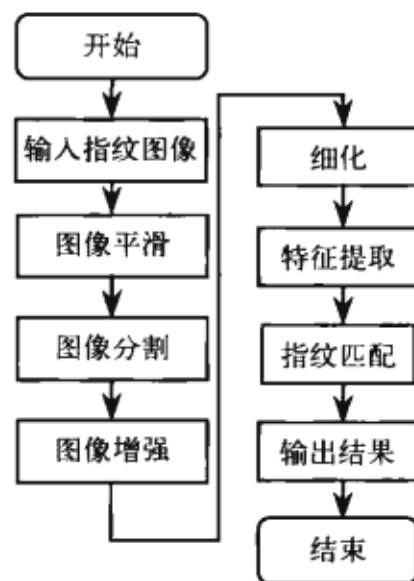


图 10-20 指纹提取与识别系统流程图

## 10.4 编程实现

指纹提取与识别系统采用 VC 2008 开发平台编程实现。从软件重用的角度考虑，在程序中调用了 FP.dll 和 FPEngine.dll 两个动态链接库。其中，FPEngine.dll 库文件实现的是指纹特征提取及匹配对比部分，而指纹图像预处理则是在 FP.dll 库文件中实现的。在调用时，FP 模块通过调用 FPEngine 模块最终实现指纹图像预处理、特征提取及指纹图像匹配整个过程。而 FPA.H 则声明了 FP.dll 函数库中的所有接口函数，最终将所有具体功能函数连接起来从而实现指纹提取及识别系统。

### 10.4.1 指纹图像分割模块

首先对方向场幅值进行平滑滤波去除噪声，然后依次对边界点以外的各个点进行判断，与阈值进行大小比较以区分是指纹本身还是背景区域，从而实现指纹图像分割。



```

////////////////////////////////////
//图像分割
////////////////////////////////////
int segment(BYTE *g_lpDivide, BYTE *g_lpTemp, int r, int threshold, int IMGW, int IMGH)
{
    // r: [in] 对幅值图像高度平滑滤波的滤波器半径
    // threshold: [in] 分割的阈值
    int x, y;
    int num = 0;
    // 对方向场幅值图像进行高度平滑滤波
    smooth(g_lpTemp, g_lpDivide, IMGW, IMGH, r, 2);
    // 图像边缘均设置为背景
    for(y = 0; y < IMGH; y++)
    {
        *(g_lpDivide + y*IMGW) = 255;
        *(g_lpDivide + y*IMGW + IMGW - 1) = 255;
    }
    for(x = 0; x < IMGW; x++)
    {
        *(g_lpDivide + x) = 255;
        *(g_lpDivide + (IMGH-1)*IMGW + x) = 255;
    }
    for(y = 1; y < IMGH-1; y++)
    {
        for(x = 1; x < IMGW-1; x++)
        {
            // 根据幅值与阈值大小判断是否为背景区域
            if(*(g_lpDivide + y*IMGW + x) < threshold)
            {
                *(g_lpDivide + y*IMGW + x) = 0;
            }
            else
            {
                *(g_lpDivide + y*IMGW + x) = 255;
                num++;
            }
        }
    }
    // 如果前景区域面积小于总面积的十分之一, 则表示前景区域太小, 返回错误
    if(num < IMGH * IMGW/10)
        return 1;
    else
        return 0;
}

```

### 10.4.2 指纹图像增强模块

指纹图像增强模块通过调用 DDIndex(\*lpDir)函数, 分别获得图像纹线方向和纹线垂直方向的 Gabor 小波模板索引, 然后利用平滑滤波器和锐化滤波器对两个方向分别进行增强运算。从而实现整个指纹图像的增强效果。

```

////////////////////////////////////
//图像增强
////////////////////////////////////
void orientEnhance(BYTE *g_lpOrient, BYTE *g_lpOrgFinger, int IMGW, int IMGH)
{
    int x, y;
    int i;
    int d = 0;
    int sum = 0;
    // 纹线方向上进行平滑滤波的平滑滤波器
    int Hw[7] = {1, 1, 1, 1, 1, 1, 1};
    // 纹线方向的垂直方向上进行锐化滤波的锐化滤波器
    int Vw[7] = {-3, -1, 3, 9, 3, -1, -3};
    int hsum = 0;
    int vsum = 0;
    int temp = 0;
    BYTE *lpSrc = NULL;
    BYTE *lpDir = NULL;
    BYTE *g_lpTemp = new BYTE[IMGW * IMGH];
    // 纹线方向上进行平滑滤波
    temp = 0;
    for(y = 0; y < IMGH; y++)
    {
        for(x = 0; x < IMGW; x++)
        {
            lpDir = g_lpOrient + temp + x;
            lpSrc = g_lpOrgFinger + temp + x;
            // 纹线方向的索引
            d = DDIndex(*lpDir);
            sum = 0;
            hsum = 0;
            for(i = 0; i < 7; i++)
            {
                if(y+g_DDSSite[d][i][1] < 0 || y+g_DDSSite[d][i][1] >= IMGH ||
                    x+g_DDSSite[d][i][0] < 0 || x+g_DDSSite[d][i][0] >= IMGW)
                {
                    continue;
                }
                sum += Hw[i]*(*(lpSrc + g_DDSSite[d][i][1]*IMGW + g_DDSSite

```

```

        [d][i][0]));
        hsum += Hw[i];
    }
    if(hsum != 0)
    {
        *(g_lpTemp + temp + x) = (BYTE)(sum/hsum);
    }
    else
    {
        *(g_lpTemp + temp + x) = 255;
    }
}
temp += IMGW;
// 纹线方向的垂直方向上进行锐化滤波
temp = 0;
for(y = 0; y < IMGH; y++)
{
    for(x = 0; x < IMGW; x++)
    {
        lpDir = g_lpOrient + temp + x;
        lpSrc = g_lpTemp + temp + x;

        // 纹线方向的垂直方向的索引
        d = (DDIndex(*lpDir)+6) % 12;
        sum = 0;
        vsum = 0;
        for(i = 0; i < 7; i++)
        {
            if(y+g_DDSite[d][i][1] < 0 || y+g_DDSite[d][i][1] >= IMGH ||
                x+g_DDSite[d][i][0] < 0 || x+g_DDSite[d][i][0] >= IMGW)
            {
                continue;
            }
            sum += Vw[i]*(*(lpSrc+g_DDSite[d][i][1]*IMGW+g_DDSite[d][i][0]));
            vsum += Vw[i];
        }
        if(vsum > 0)
        {
            sum /= vsum;
            if(sum > 255)
            {
                *(g_lpOrgFinger + temp + x) = 255;
            }
            else if(sum < 0)
            {
                *(g_lpOrgFinger + temp + x) = 0;
            }
        }
    }
}

```

```

        else
        {
            *(g_lpOrgFinger + temp + x) = (BYTE)sum;
        }
    }
    else
    {
        *(g_lpOrgFinger + temp + x) = 255;
    }
}
temp += IMGW;
}
}

```

### 10.4.3 指纹图像二值化模块

指纹图像进行二值化分几个小步骤，首先获取某坐标点传入的纹线方向的 Gabor 小波模板索引，然后对该点按纹线方向和纹线垂直方向分别使用 Gabor 小波简化模板进行二值化运算，最终转变为高质量的黑白图像。

```

////////////////////////////////////
//二值化
////////////////////////////////////
int binary(BYTE *g_lpOrgFinger, BYTE *g_lpTemp, BYTE *g_lpOrient, int IMGW, int IMGH)
{
    int x, y;
    int i;
    int d = 0;
    int sum = 0;
    // 纹线方向上的 7 个点的权值
    int Hw[7] = {2, 2, 3, 4, 3, 2, 2};
    // 纹线方向的垂直方向上的 7 个点的权值
    int Vw[7] = {1, 1, 1, 1, 1, 1, 1};
    int hsum = 0; // 纹线方向上的 7 个点的加权和
    int vsum = 0; // 纹线方向的垂直方向上的 7 个点的加权和
    int Hv = 0; // 纹线方向上的 7 个点的加权平均值
    int Vv = 0; // 纹线方向的垂直方向上的 7 个点的加权平均值
    int temp = 0;
    BYTE *lpSrc = NULL; // 指纹图像像素点指针
    BYTE *lpDir = NULL; // 纹线方向指针
    temp = 0;
    for(y = 0; y < IMGH; y++)
    {
        for(x = 0; x < IMGW; x++)
        {
            lpDir = g_lpOrient + temp + x;

```

```

lpSrc = g_lpOrgFinger + temp + x;
// 如果该点非常黑, 则在临时缓冲区内置该点为黑点, 值为 0
if(*lpSrc < 4)
{
    *(g_lpTemp + temp + x) = 0;
    continue;
}
// 计算方向索引 (量化为 12 个方向)
d = DDIndex(*lpDir);

// 计算当前点在纹线方向上的加权平均值
sum = 0;
hsum = 0;
for(i = 0; i < 7; i++)
{
    // 坐标是否越界
    if(y+g_DDSite[d][i][1] < 0 || y+g_DDSite[d][i][1] >= IMGH ||
        x+g_DDSite[d][i][0] < 0 || x+g_DDSite[d][i][0] >= IMGW)
    {
        continue;
    }
    sum += Hw[i] * (*(lpSrc + g_DDSite[d][i][1]*IMGW + g_DDSite[d][i][0]));
    hsum += Hw[i];
}
if(hsum != 0)
{
    Hv = sum/hsum;
}
else
{
    Hv = 255;
}
// 纹线方向的垂直方向的索引
d = (d+6)%12;
// 计算当前点在纹线方向的垂直方向上的加权平均值
sum = 0;
vsum = 0;
for(i = 0; i < 7; i++)
{
    if(y+g_DDSite[d][i][1] < 0 || y+g_DDSite[d][i][1] >= IMGH ||
        x+g_DDSite[d][i][0] < 0 || x+g_DDSite[d][i][0] >= IMGW)
    {
        continue;
    }
    sum += Vw[i] * (*(lpSrc + g_DDSite[d][i][1]*IMGW + g_DDSite[d][i][0]));
    vsum += Vw[i];
}

```



```

    }
    if(vsum != 0)
    {
        Vv = sum/vsum;
    }
    else
    {
        Vv = 255;
    }
    if(Hv < Vv)
    {
        // 纹线方向上加权平均值较小则置当前点为黑点
        *(g_lpTemp + temp + x) = 0;
    }
    else
    {
        // 纹线方向上加权平均值较大则置当前点为白点
        *(g_lpTemp + temp + x) = 255;
    }
}
temp += IMGW;
}
// 将临时缓冲区内数据复制到原始图像数据缓冲区
//memcpy((void *)g_lpOrgFinger, (void *)g_lpTemp, IMGSIZE);
return 0;
}

```

#### 10.4.4 细化模块

按照 10.1.3 中给出的细化方法，细化模块通过对整个图像的水平 and 垂直两个方向进行运算处理，最终输出细化结果得到一像素宽的黑白图像。

```

////////////////////////////////////
//细化处理
////////////////////////////////////
int imageThin(BYTE *lpBits, BYTE *g_lpTemp, int Width, int Height)
{
    // lpBits: [in, out] 要细化的图像数据缓冲区
    // Width: [in] 要细化的图像宽度
    // Height: [in] 要细化的图像高度
    BYTE erasetable[256]={
        0,0,1,1,0,0,1,1,        1,1,0,1,1,1,0,1,
        1,1,0,0,1,1,1,1,        0,0,0,0,0,0,0,1,
        0,0,1,1,0,0,1,1,        1,1,0,1,1,1,0,1,
        1,1,0,0,1,1,1,1,        0,0,0,0,0,0,0,1,
        1,1,0,0,1,1,0,0,        0,0,0,0,0,0,0,0,
    }
}

```



```

0,0,0,0,0,0,0,0,0,0,    0,0,0,0,0,0,0,0,0,0,
1,1,0,0,1,1,0,0,0,0,    1,1,0,1,1,1,0,1,
0,0,0,0,0,0,0,0,0,0,    0,0,0,0,0,0,0,0,0,0,
0,0,1,1,0,0,1,1,1,1,    1,1,0,1,1,1,0,1,
1,1,0,0,1,1,1,1,1,1,    0,0,0,0,0,0,0,0,1,
0,0,1,1,0,0,1,1,1,1,    1,1,0,1,1,1,0,1,
1,1,0,0,1,1,1,1,1,1,    0,0,0,0,0,0,0,0,0,0,
1,1,0,0,1,1,0,0,0,0,    0,0,0,0,0,0,0,0,0,0,
1,1,0,0,1,1,1,1,1,1,    0,0,0,0,0,0,0,0,0,0,
1,1,0,0,1,1,0,0,0,0,    1,1,0,1,1,1,0,0,0,
1,1,0,0,1,1,1,1,1,1,    1,1,0,0,1,0,0,0,
1,1,0,0,1,1,1,0,0,0,
};
int      x,y;
int      num;
BOOL      Finished;
BYTE      nw,n,ne,w,e,sw,s,se;
BYTE      *lpPtr = NULL;
BYTE      *lpTempPtr = NULL;
memcpy((void *)g_lpTemp, (void *)lpBits, Width*Height);

//结束标志置成假
Finished=FALSE;
while(!Finished){ //还没有结束
    //结束标志置成假
    Finished=TRUE;
    //先进行水平方向的细化
    for (y=1;y<Height-1;y++)
    { //注意为防止越界, y 的范围从 1 到高度-2
        //lpPtr 指向原图数据, lpTempPtr 指向新图数据
        lpPtr=(BYTE *)lpBits+y*Width;
        lpTempPtr=(BYTE *)g_lpTemp+y*Width;
        x=1; //注意为防止越界, x 的范围从 1 到宽度-2
        while(x<Width-1)
        {
            if(*(lpPtr+x)==0)
            { //是黑点才做处理
                w = *(lpPtr+x-1); //左邻点
                e = *(lpPtr+x+1); //右邻点
                if( (w==255)|| (e==255)){
                    //如果左右两个邻居中至少有一个是白点才处理
                    nw=*(lpPtr+x+Width-1); //左上邻点
                    n=*(lpPtr+x+Width); //上邻点
                    ne=*(lpPtr+x+Width+1); //右上邻点
                    sw=*(lpPtr+x-Width-1); //左下邻点
                    s=*(lpPtr+x-Width); //下邻点
                    se=*(lpPtr+x-Width+1); //右下邻点
                    //计算索引

```

```

        num=nw/255+n/255*2+ne/255*4+w/255*8+e/255*16+
            sw/255*32+s/255*64+se/255*128;
        if(erasetable[num]==1){ //经查表, 可以删除
            //在原图缓冲区中将该黑点删除
            *(lpPtr+x)=255;
            //结果图中该黑点也删除
            *(lpTempPtr+x)=255;
            Finished=FALSE; //有改动, 结束标志置成假
            x++; //水平方向跳过一个像素
        }
    }
    x++; //扫描下一个像素
}

//再进行垂直方向的细化
for (x=1;x<Width-1;x++){ //注意为防止越界, x 的范围从 1 到宽度-2
    y=1; //注意为防止越界, y 的范围从 1 到高度-2
    while(y<Height-1){
        lpPtr=lpBits+y*Width;
        lpTempPtr=g_lpTemp+y*Width;
        if(*(lpPtr+x)==0){ //是黑点才做处理
            n=*(lpPtr+x+Width);
            s=*(lpPtr+x-Width);
            if( (n==255)|| (s==255)){
                //如果上下两个邻居中至少有一个是白点才处理
                nw=*(lpPtr+x+Width-1);
                ne=*(lpPtr+x+Width+1);
                w=*(lpPtr+x-1);
                e=*(lpPtr+x+1);
                sw=*(lpPtr+x-Width-1);
                se=*(lpPtr+x-Width+1);
                //计算索引
                num=nw/255+n/255*2+ne/255*4+w/255*8+e/255*16+
                    sw/255*32+s/255*64+se/255*128;
                if(erasetable[num]==1){ //经查表, 可以删除
                    // 在原图缓冲区中将该黑点删除
                    *(lpPtr+x)=255;
                    //结果图中该黑点也删除
                    *(lpTempPtr+x)=255;
                    Finished=FALSE; //有改动, 结束标志置成假
                    y++; //垂直方向跳过一个像素
                }
            }
        }
        y++; //扫描下一个像素
    }
}

```

```

    }
}

//memcpy((void *)lpBits, (void *)g_lpTemp, Width*Height);
return 0;
}

```

### 10.4.5 特征点提取模块

端点和分叉点的提取程序较为简单，找到当前点及其周围 8 个点的位置，根据八邻域法和前文提到的判断规则在除边界点外的其他点搜索以找到端点和分叉点。奇异点（即中心点和三角点）需要借助方向场来进行计算和判断。

```

////////////////////////////////////
//  IsFork: 判断某点是否为分叉点
////////////////////////////////////
bool  IsFork(BYTE *lpNow)
{
    //lpNow : [in] 当前点的地址
    int i, sum;
    // 某点周围 8 个点的地址偏移
    int SiteD8[8] = {IMGW-1, IMGW, IMGW+1, 1, -IMGW+1, -IMGW, -IMGW-1, -1};

    // 8 个点所有相邻两个点的差的绝对值的和如果为 6*255 则为分叉点
    sum = 0;
    for(i = 0; i < 8; i++)
    {
        sum += abs(*(lpNow + SiteD8[(i+1)%8]) - *(lpNow + SiteD8[i]));
    }
    if(sum == 255*6)
    {
        return true;
    }
    else
    {
        return false;
    }
}

////////////////////////////////////
//  IsEnd: 判断某点是否为端点
////////////////////////////////////
bool  IsEnd(BYTE *lpNow)
{
    // lpNow : [in] 当前点的地址
    int i, sum;

```

```

// 某点周围 8 个点的地址偏移
int SiteD8[8] = {IMGW-1, IMGW, IMGW+1, 1, -IMGW+1, -IMGW, -IMGW-1, -1};

// 8 个点所有相邻两个点的差的绝对值的和如果为 2*255 则为端点
sum = 0;
for(i = 0; i < 8; i++)
{
    sum += abs(*(lpNow + SiteD8[(i+1)%8]) - *(lpNow + SiteD8[i]));
}
if(sum == 255*2)
{
    return true;
}
else
{
    return false;
}
}

/////////////////////////////////////////////////////////////////
//提取中心点和三角点
/////////////////////////////////////////////////////////////////
int getSingular(BYTE *lpOrient, int Width, int Height, int *lpNum, PPOINT lpArr,
                char flag)
{
    int    x, y, i;
    int    num;
    int    sum1, sum2;
    int    d;
    unsigned char *pOriMap;
    int    oriV;
    int    a1, a2;
    DBLPOINT singuArr[30];
    int    value;
    double dis;
    bool    bFound = false;
    bool    fg = false;
    int    D3[8][2] = {
        -1,-1, -1,0, -1,1, 0,1, 1,1, 1,0, 1,-1, 0,-1
    };
    int    D5[12][2] = {
        -2,-1, -2,0, -2,1, -1,2, 0,2, 1,2, 2,1, 2,0, 2,-1, 1,-2, 0,-2, -1,-2
    };
    int    D7[24][2] = {
        -3,-3, -3,-2, -3,-1, -3,0, -3,1, -3,2, -3,3, -2,3, -1,3, 0,3, 1,3, 2,3,
        3,3, 3,2, 3,1, 3,0, 3,-1, 3,-2, 3,-3, 2,-3, 1,-3, 0,-3, -1,-3, -2,-3
    };
    num = 0;

```

```

memset(singuArr, 0, sizeof(singuArr));           //初始化奇异点数组, 清零
for(y = 3; y < Height-3; y++)
{
    for(x = 3; x < Width-3; x++)                 //遍历整幅图
    {
        pOriMap = lpOrient + y*Width + x;        //获得某点的方向场指针
        oriV = *pOriMap;                         //获取某点的方向场的值
        if(oriV == 255)                          //若是背景区域, 则跳入下一个循环
        {
            continue;
        }
        fg = false;
        for(i = 0; i < 24; i++)
        {
            if(pOriMap[D7[i][1]*Width + D7[i][0]]==255)
            {
                fg = true;
                break;
            }
        }
        if(fg)
        {
            continue;
        }
        sum1 = 0;
        for(i = 0; i < 8; i++)
        {
            a1 = pOriMap[D3[i][1]*Width + D3[i][0]]/24;
            a2 = pOriMap[D3[(i+1)%8][1]*Width + D3[(i+1)%8][0]]/24;
            d = getOriChange(a1, a2, flag);

            if(abs(d) > 5)
            {
                break;
            }
            sum1 += d;
        }
        sum2 = 0;
        for(i = 0; i < 12; i++)
        {
            a1 = pOriMap[D5[i][1]*Width + D5[i][0]]/24;
            a2 = pOriMap[D5[(i+1)%12][1]*Width + D5[(i+1)%12][0]]/24;
            d = getOriChange(a1, a2, flag);

            if(abs(d) > 5)
            {
                break;
            }
        }
    }
}

```

```

        }

        sum2 += d;
    }
    if(flag == -1)    //中心点
    {
        value = -10;
    }
    else if(flag == 1)    //三角点
    {
        value = 10;
    }
    if(sum2 == value && sum1 == value)
    {
        bFound = false;
        for(i = 0; i < num; i++)
        {
            dis = sqrt((double)((x - singuArr[i].x)*(x - singuArr[i].x) +
                (y - singuArr[i].y)*(y - singuArr[i].y)));
            if(dis < 4)
            {
                singuArr[i].x = (singuArr[i].x + x)/2.0;
                singuArr[i].y = (singuArr[i].y + y)/2.0;
                bFound = true;
                break;
            }
        }
        if(!bFound)
        {
            singuArr[num].x = x;
            singuArr[num].y = y;
            num++;
            //若奇异点数量超过最大限制, 则停止寻找奇异点
            if(num >= MAX_SINGULARYNUM)
            {
                goto next;
            }
        }
    }
}

next:
*lpNum = num;    //奇异点的个数
for(i = 0; i < num; i++) //将找到的奇异点存入输入数组
{
    lpArr[i].x = (long)singuArr[i].x;
    lpArr[i].y = (long)singuArr[i].y;
}

```



```

    }
    return 0;
}

```

### 10.4.6 指纹图像比对模块

在指纹图像比对模块中，首先要进行坐标转换，对两个对比图像进行图像配准，然后根据特征点进行全局比对，全局比对中又包括中心点的配准和匹配、中心区域的配准和匹配、全局配准和匹配，最终得到匹配结果，判断出是相同的指纹还是不同的指纹。

```

////////////////////////////////////
//align: 将指纹特征按一定的角度和位置偏移进行坐标变换
////////////////////////////////////
void align(FEATUREPTR lpFeature, FEATUREPTR lpAlignedFeature, MINUTIAPTR
lpFeatureCore, int rotation, int transx, int transy)
{
    ///////////////////////////////////
    // lpFeature:[in] 要变换的指纹特征
    // lpAlignedFeature:[out] 进行坐标变换后的指纹特征
    // lpFeatureCore:[in] 旋转变换的中心特征点
    // rotation:[in] 旋转角度
    // transx: [in] 水平偏移
    // transy: [in] 垂直偏移
    ///////////////////////////////////
    int i;
    int x, y;
    int cx, cy;
    double rota, sinv, cosv;
    // 复制整个结构信息
    *lpAlignedFeature = *lpFeature;
    // 坐标转换的中心点坐标
    cx = lpFeatureCore->x;
    cy = lpFeatureCore->y;
    // 旋转的弧度
    rota = rotation/EPI;
    // 旋转弧度的 sin 值
    sinv = sin(rota);
    // 旋转弧度的 cos 值
    cosv = cos(rota);
    for(i = 0; i < lpFeature->MinutiaNum; i++)
    {
        x = lpFeature->MinutiaArr[i].x;
        y = lpFeature->MinutiaArr[i].y;
        // 坐标转换后的新坐标
        lpAlignedFeature->MinutiaArr[i].x = (int)(cx + cosv*(x-cx) - sinv*(y-cy)
+ transx + 0.5);
    }
}

```

```

        lpAlignedFeature->MinutiaArr[i].y = (int)(cy + sinv*(x-cx) + cosv*(y-cy)
        + transy + 0.5);
        // 旋转后特征点的新方向
        lpAlignedFeature->MinutiaArr[i].Direction = (lpFeature->MinutiaArr[i].
        Direction + rotation) % 360;
    }
}
// alignMatch: 比对两个坐标系对齐的指纹特征
void alignMatch(FEATUREPTR lpAlignFeature, FEATUREPTR lpTemplate,
                PMATCHRESULT lpMatchResult, VF_FLAG matchMode)
{
    int i, j;
    charflagA[MAXMINUTIANUM]; // 标记 lpAlignFeature 中的特征点是否已经有匹配对象
    charflagT[MAXMINUTIANUM]; // 标记 lpTemplate 中的特征点是否已经有匹配对象
    int x1, y1, x2, y2;
    int dis, angle;
    int score, matchNum, s;
    int num1, num2;
    int step = 1;
    num1 = lpAlignFeature->MinutiaNum;
    num2 = lpTemplate->MinutiaNum;
    // 标记清零
    memset(flagA, 0, MAXMINUTIANUM);
    memset(flagT, 0, MAXMINUTIANUM);
    score = 0; // 总分清零
    matchNum = 0; // 匹配特征点数清零
    // 相同类型特征点的匹配分数
    for(i = 0; i < lpTemplate->MinutiaNum; i++)
    {
        if(flagT[i]) // 是否已有匹配对象
            continue;
        for(j = 0; j < lpAlignFeature->MinutiaNum; j++)
        {
            if(flagA[j]) // 是否已有匹配对象
                continue;
            // 特征点类型是否相同
            if(lpTemplate->MinutiaArr[i].Type != lpAlignFeature->MinutiaArr[j].
            Type)
                continue;
            // 特征点方向夹角
            angle = AngleAbs360(lpTemplate->MinutiaArr[i].Direction,
            lpAlignFeature->MinutiaArr[j].Direction);
            // 夹角>=10 则不匹配
            if(angle >= 10)
                continue;

```

```

x1 = lpTemplate->MinutiaArr[i].x;
y1 = lpTemplate->MinutiaArr[i].y;
x2 = lpAlignFeature->MinutiaArr[j].x;
y2 = lpAlignFeature->MinutiaArr[j].y;
// 水平距离>=10 则不匹配
if(abs(x1-x2) >= 10)
    continue;
// 垂直距离>=10 则不匹配
if(abs(y1-y2) >= 10)
    continue;
// 两特征点间的距离
dis = DisTbl[abs(y1-y2)][abs(x1-x2)];
// 距离>=10 则不匹配
if(dis >= 10)
    continue;
// 对这两个特征点做标记, 表示已经有匹配对象
flagA[j] = 1;
flagT[i] = 1;
// 总分加上这两个特征点的匹配分数
// 此表明 dis 和 angle 越大, 分数越小
score += (10-angle);
score += (10-dis);
// 匹配特征点数加一
matchNum++;

// 如果是快速比对模式
if(matchMode == VF_MATCHMODE_IDENTIFY && matchNum >= 8)
{
    // 计算相似度
    s = 4 * score * matchNum * MAXMINUTIANUM / ((num1 + num2) * (num1
                                                    + num2));

    if(s > 100) // 相似度足够大则返回比对结果
    {
        lpMatchResult->MMCount = matchNum;
        lpMatchResult->Rotation = 0;
        lpMatchResult->Similarity = s;
        lpMatchResult->TransX = 0;
        lpMatchResult->TransY = 0;
        return;
    }
}
}

if(matchMode != VF_MATCHMODE_IDENTIFY)
{
    // 图像处理的误差可能导致将端点处理成叉点或叉点处理成端点, 假设概率
    // 为 50%, 计算此种情况的分数

```

```

for(i = 0; i < lpTemplate->MinutiaNum; i++)
{
    if(flagT[i]) // 是否已有匹配对象
        continue;
    for(j = 0; j < lpAlignFeature->MinutiaNum; j++)
    {
        if(flagA[j]) // 是否已有匹配对象
            continue;
        // 是否类型不同

        if(lpTemplate->MinutiaArr[i].Type==lpAlignFeature->MinutiaArr[j].
Type)
            continue;
        // 特征点方向夹角
        angle = AngleAbs360(lpTemplate->MinutiaArr[i].Direction,
            lpAlignFeature->MinutiaArr[j].Direction);
        // 夹角>=10 则不匹配
        if(angle >= 10)
            continue;
        x1 = lpTemplate->MinutiaArr[i].x;
        y1 = lpTemplate->MinutiaArr[i].y;
        x2 = lpAlignFeature->MinutiaArr[j].x;
        y2 = lpAlignFeature->MinutiaArr[j].y;

        // 水平距离>=10 则不匹配
        if(abs(x1-x2) >= 10)
            continue;
        // 垂直距离>=10 则不匹配
        if(abs(y1-y2) >= 10)
            continue;
        // 两特征点间的距离
        dis = DisTbl[abs(y1-y2)][abs(x1-x2)];
        // 距离>=10 则不匹配
        if(dis >= 10)
            continue;
        // 对这两个特征点做标记, 表示已经有匹配对象
        flagA[j] = 1;
        flagT[i] = 1;
        // 总分加上这两个特征点的匹配分数
        score += ((10-angle)/2);
        score += ((10-dis)/2);
        // 匹配特征点数加一
        matchNum++;
    }
}
// 计算相似度, 返回比对结果

```

```

    s = 4 * score * matchNum * MAXMINUTIANUM / ((num1+num2)*(num1+num2));
    lpMatchResult->MMCount = matchNum;
    lpMatchResult->Rotation = 0;
    lpMatchResult->Similarity = s;
    lpMatchResult->TransX = 0;
    lpMatchResult->TransY = 0;
}
////////////////////////////////////
//中心点的配准及匹配
////////////////////////////////////
void coreMatch(FEATUREPTR lpFeature, FEATUREPTR lpTemplate,
               PMATCHRESULT lpMatchResult, VF_FLAG matchMode, int n, int m)
{
    MATCHRESULT alignMax;    // 相似度最大的比对结果
    MATCHRESULT globalMatchResult;    // 比对结果
    int agate = 8;          // 三角拓扑结构角度误差
    int num = 0;
    // 初始化最好的比对结果
    alignMax.Similarity = 0;
    alignMax.MMCount = 0;
    alignMax.Rotation = 0;
    alignMax.TransX = 0;
    alignMax.TransY = 0;
    FEATURE alignFeature;    // 对齐后的指纹特征
    // 位置偏移
    int transx = (lpTemplate->MinutiaArr[n].x - lpFeature->MinutiaArr[m].x);
    int transy = (lpTemplate->MinutiaArr[n].y - lpFeature->MinutiaArr[m].y);
    for(int i = 0; i < lpFeature->MinutiaNum; i++)
    {
        for(int j = 0; j < lpTemplate->MinutiaNum; j++)
        {
            alignFeature.MinutiaNum = 0;

            if (lpFeature->MinutiaArr[i].Type == VF_MINUTIA_CORE || lpTemplate->
                MinutiaArr[j].Type == VF_MINUTIA_CORE) continue;
            if (lpFeature->MinutiaArr[i].Type == VF_MINUTIA_DELTA || lpTemplate
                ->MinutiaArr[j].Type == VF_MINUTIA_DELTA) continue;
            int rotation = GetAngle(lpTemplate->MinutiaArr[j].x, lpTemplate
                ->MinutiaArr[j].y,
                lpFeature->MinutiaArr[i].x, lpFeature->MinutiaArr[i].y);
            align(lpFeature, &alignFeature, &lpFeature->MinutiaArr[i], rotation,
                transx, transy);    // 将两个对齐的指纹特征进行比对
            alignMatch(&alignFeature, lpTemplate, &globalMatchResult, matchMode);
            // 如果比对结果比最好的比对结果更好, 则更新 alignMax
            if(globalMatchResult.Similarity > alignMax.Similarity)
            {
                alignMax.MMCount = globalMatchResult.MMCount;
            }
        }
    }
}

```



```

alignMax.Similarity = globalMatchResult.Similarity;
alignMax.Rotation = rotation;
alignMax.TransX = transx;
alignMax.TransY = transy;

// 如果是快速比对模式, 在相似度达到一定程度后则退出
if(matchMode==VF_MATCHMODE_IDENTIFY&& alignMax.MMCount >= 8 )
{
    if(alignMax.Similarity > 100)
    {
        *lpMatchResult = alignMax;
        return;
    }
}

}

}

//最终比对结果
*lpMatchResult = alignMax;
}

////////////////////////////////////
//三角点的配准及匹配
////////////////////////////////////
void deltaMatch(FEATUREPTR lpFeature, FEATUREPTR lpTemplate,
    PMATCHRESULT lpMatchResult, VF_FLAG matchMode, vector<int>& n_delta, vector
<int>& m_delta)
{
    MATCHRESULT alignMax;    // 相似度最大的比对结果
    MATCHRESULT globalMatchResult;    // 比对结果
    int agate = 8;    // 三角拓扑结构角度误差
    int num = 0;
    // 初始化最好的比对结果
    alignMax.Similarity = 0;
    alignMax.MMCount = 0;
    alignMax.Rotation = 0;
    alignMax.TransX = 0;
    alignMax.TransY = 0;
    FEATURE alignFeature;    // 对齐后的指纹特征
    int n, m;
    for(int nn = 0; nn < n_delta.size(); nn++)
        for(int mm = 0; mm < m_delta.size(); mm++)
        {
            n = (int)n_delta[nn];
            m = (int)m_delta[mm];
            // 位置偏移

```



```

int transx = (lpTemplate->MinutiaArr[n].x - lpFeature->MinutiaArr[m].x);
int transy = (lpTemplate->MinutiaArr[n].y - lpFeature->MinutiaArr[m].y);
for(int i = 0; i < lpFeature->MinutiaNum; i++)
{
    for(int j = 0; j < lpTemplate->MinutiaNum; j++)
    {
        alignFeature.MinutiaNum = 0;

        if (lpFeature->MinutiaArr[i].Type == VF_MINUTIA_CORE ||
lpTemplate->MinutiaArr[j].Type == VF_MINUTIA_CORE) continue;
        if(lpFeature->MinutiaArr[i].Type== VF_MINUTIA_DELTA ||
lpTemplate->MinutiaArr[j].Type == VF_MINUTIA_DELTA)
            continue;
        int rotation=GetAngle(lpFeature->MinutiaArr[i].x,
lpFeature->MinutiaArr[i].y, lpTemplate->MinutiaArr[j].x, lpTemplate->MinutiaArr[j].y);
        align(lpFeature, &alignFeature, &lpFeature->MinutiaArr[i],
rotation, transx, transy);

        // 将两个对齐的指纹特征进行比对
        alignMatch(&alignFeature, lpTemplate, &globalMatchResult, matchMode);
        // 如果比对结果比最好的比对结果更好, 则更新 alignMax
        if(globalMatchResult.Similarity > alignMax.Similarity)
        {
            alignMax.MMCount = globalMatchResult.MMCount;
            alignMax.Similarity = globalMatchResult.Similarity;
            alignMax.Rotation = rotation;
            alignMax.TransX = transx;
            alignMax.TransY = transy;

            // 如果是快速比对模式, 在相似度达到一定程度后则退出
            if(matchMode==VF_MATCHMODE_IDENTIFY&& alignMax.MMCount >= 8 )
            {
                if(alignMax.Similarity > 100)
                {
                    *lpMatchResult = alignMax;
                    return;
                }
            }
        }
    }
}

//最终比对结果
*lpMatchResult = alignMax;
}

//////////////////////////
//求距离
//////////////////////////

```

```

int dist(int x0, int y0, int x1, int y1)
{
    return (int)sqrt((float)(x0 - x1) * (float)(x0 - x1) + (float)(y0 - y1) * (float)(y0 - y1));
}
////////////////////////////////////
//中心区域配准及匹配
////////////////////////////////////
void centralMatch(FEATUREPTR lpFeature, FEATUREPTR lpTemplate, PMATCHRESULT
lpMatchResult, VF_FLAG matchMode)
{
    // lpFeature: [in] 要比对的第一个指纹特征指针
    // lpTemplate: [in] 要比对的第二个指纹特征指针
    // lpMatchResult: [out] 比对结果指针
    // matchMode: [in] 比对模式
    int m, n, a1, a2;
    int rotation; // 旋转角度
    int transx, transy; // 位置偏移
    FEATURE alignFeature; // 对齐后的指纹特征
    MATCHRESULT alignMax; // 相似度最大的比对结果
    MATCHRESULT globalMatchResult; // 比对结果
    int agate = 8; // 三角拓扑结构角度误差
    int num = 0;
    // 初始化最好的比对结果
    alignMax.Similarity = 0;
    alignMax.MMCount = 0;
    alignMax.Rotation = 0;
    alignMax.TransX = 0;
    alignMax.TransY = 0;
    int nx = 0, ny = 0;
    for(n = 0; n < lpTemplate->MinutiaNum; n++)
    {
        nx += lpTemplate->MinutiaArr[n].x;
        ny += lpTemplate->MinutiaArr[n].y;
    }
    nx = nx / lpTemplate->MinutiaNum;
    ny = ny / lpTemplate->MinutiaNum;
    int mx = 0, my = 0;
    for(m = 0; m < lpFeature->MinutiaNum; m++)
    {
        mx += lpFeature->MinutiaArr[m].x;
        my += lpFeature->MinutiaArr[m].y;
    }
    mx = mx / lpFeature->MinutiaNum;
    my = my / lpFeature->MinutiaNum;
    int Counter = 0;
    // 对相同类型的指纹特征两两作为同一个指纹特征进行对齐比对

```

```

for(n = 0; n < lpTemplate->MinutiaNum; n++)
{
    if(dist(nx, ny, lpTemplate->MinutiaArr[n].x, lpTemplate->MinutiaArr[n].y) >
        CENTRALRADIUS) continue;
    for(m = 0; m < lpFeature->MinutiaNum; m++)
    {
        // 不同类型则不对
        if(lpFeature->MinutiaArr[m].Type != lpTemplate->MinutiaArr[n].Type)
            continue;
        if(dist(mx, my, lpFeature->MinutiaArr[m].x,
            lpFeature->MinutiaArr[m].y) > CENTRALRADIUS) continue;
        Counter++;
        if(matchMode == VF_MATCHMODE_IDENTIFY)
        {
            // 特征点三角拓扑结构比对, 相似则进行配准
            if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
                MinutiaArr[n].Triangle[0] != 255)
            {
                a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
                    lpFeature->MinutiaArr[m].Direction % 180);
                a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
                    lpTemplate->MinutiaArr[n].Direction % 180);
                if(abs(a1-a2)>agate)
                    continue;
            }
            if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
                MinutiaArr[n].Triangle[0] != 255 &&
                lpFeature->MinutiaArr[m].Triangle[1] != 255 && lpTemplate->
                MinutiaArr[n].Triangle[1] != 255)
            {
                a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
                    lpFeature->MinutiaArr[m].Triangle[1]);
                a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
                    lpTemplate->MinutiaArr[n].Triangle[1]);
                if(abs(a1-a2)>agate)
                    continue;
            }
            if(lpFeature->MinutiaArr[m].Triangle[2] != 255 && lpTemplate
                ->MinutiaArr[n].Triangle[2] != 255 &&
                lpFeature->MinutiaArr[m].Triangle[1] != 255 && lpTemplate->
                MinutiaArr[n].Triangle[1] != 255)
            {
                a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[1],
                    lpFeature->MinutiaArr[m].Triangle[2]);
                a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[1],
                    lpTemplate->MinutiaArr[n].Triangle[2]);
                if(abs(a1-a2)>agate)

```



```

        continue;
    }
    if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[0] != 255 &&
        lpFeature->MinutiaArr[m].Triangle[2] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[2] != 255)
    {
        a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
            lpFeature->MinutiaArr[m].Triangle[2]);
        a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
            lpTemplate->MinutiaArr[n].Triangle[2]);
        if(abs(a1-a2)>agate)
            continue;
    }
}
alignFeature.MinutiaNum = 0;
// 旋转角度
rotation = GetAngleDis(lpFeature->MinutiaArr[m].Direction,
                        lpTemplate->MinutiaArr[n].Direction);
// 位置偏移
transx = (lpTemplate->MinutiaArr[n].x - lpFeature->MinutiaArr[m].x);
transy = (lpTemplate->MinutiaArr[n].y - lpFeature->MinutiaArr[m].y);
// 将 lpFeature 与 lpTemplate 对齐
align(lpFeature, &alignFeature, &lpFeature->MinutiaArr[m],
      rotation, transx, transy);
// 将两个对齐的指纹特征进行比对
alignMatch(&alignFeature, lpTemplate, &globalMatchResult, matchMode);
// 如果比对结果比最好的比对结果更好, 则更新 alignMax
if(globalMatchResult.Similarity > alignMax.Similarity)
{
    alignMax.MMCount = globalMatchResult.MMCount;
    alignMax.Similarity = globalMatchResult.Similarity;
    alignMax.Rotation = rotation;
    alignMax.TransX = transx;
    alignMax.TransY = transy;
    // 如果是快速比对模式, 在相似度达到一定程度后则退出
    if(matchMode==VF_MATCHMODE_IDENTIFY&& alignMax.MMCount >= 8 )
    {
        if(alignMax.Similarity > 100)
        {
            *lpMatchResult = alignMax;
            return;
        }
    }
}
}
}
}

```

```

        //最终比对结果
        *lpMatchResult = alignMax;
    }
    //////////////////////////////////////
    //全局配准及匹配
    //////////////////////////////////////
    void globalMatch(FEATUREPTR lpFeature, FEATUREPTR lpTemplate,
                    PMATCHRESULT lpMatchResult, VF_FLAG matchMode)
    {
        //lpFeature: [in] 要比对的第一个指纹特征指针
        //lpTemplate: [in] 要比对的第二个指纹特征指针
        //lpMatchResult: [out] 比对结果指针
        //matchMode: [in] 比对模式
        int m, n, a1, a2;
        int rotation; // 旋转角度
        int transx, transy; // 位置偏移
        FEATURE alignFeature; // 对齐后的指纹特征
        MATCHRESULT alignMax; // 相似度最大的比对结果
        MATCHRESULT globalMatchResult; // 比对结果
        int agate = 8; // 三角拓扑结构角度误差
        int num = 0;
        // 初始化最好的比对结果
        alignMax.Similarity = 0;
        alignMax.MMCount = 0;
        alignMax.Rotation = 0;
        alignMax.TransX = 0;
        alignMax.TransY = 0;
        // 对相同类型的指纹特征两两作为同一个指纹特征进行对齐比对
        for(n = 0; n < lpTemplate->MinutiaNum; n++)
        {
            for(m = 0; m < lpFeature->MinutiaNum; m++)
            {
                // 不同类型则不对比
                if(lpFeature->MinutiaArr[m].Type != lpTemplate->MinutiaArr[n].Type)
                    continue;
                if(matchMode == VF_MATCHMODE_IDENTIFY)
                {
                    // 特征点三角拓扑结构比对, 相似则进行配准
                    if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
                        MinutiaArr[n].Triangle[0] != 255)
                    {
                        a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
                            lpFeature->MinutiaArr[m].Direction % 180);
                        a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
                            lpTemplate->MinutiaArr[n].Direction % 180);
                        if(abs(a1-a2)>agate)

```

```

        continue;
    }
    if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[0] != 255 &&
        lpFeature->MinutiaArr[m].Triangle[1] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[1] != 255)
    {
        a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
            lpFeature->MinutiaArr[m].Triangle[1]);
        a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
            lpTemplate->MinutiaArr[n].Triangle[1]);
        if(abs(a1-a2)>agate)
            continue;
    }
    if(lpFeature->MinutiaArr[m].Triangle[2] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[2] != 255 &&
        lpFeature->MinutiaArr[m].Triangle[1] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[1] != 255)
    {
        a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[1],
            lpFeature->MinutiaArr[m].Triangle[2]);
        a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[1],
            lpTemplate->MinutiaArr[n].Triangle[2]);
        if(abs(a1-a2)>agate)
            continue;
    }
    if(lpFeature->MinutiaArr[m].Triangle[0] != 255 && lpTemplate->
        MinutiaArr[n].Triangle[0] != 255 &&
        lpFeature->MinutiaArr[m].Triangle[2] != 255 &&
        lpTemplate->MinutiaArr[n].Triangle[2] != 255)
    {
        a1=GetJiajiao(lpFeature->MinutiaArr[m].Triangle[0],
            lpFeature->MinutiaArr[m].Triangle[2]);
        a2=GetJiajiao(lpTemplate->MinutiaArr[n].Triangle[0],
            lpTemplate->MinutiaArr[n].Triangle[2]);
        if(abs(a1-a2)>agate)
            continue;
    }
}
alignFeature.MinutiaNum = 0;
// 旋转角度
rotation = GetAngleDis(lpFeature->MinutiaArr[m].Direction,
                        lpTemplate->MinutiaArr[n].Direction);
// 位置偏移
transx = (lpTemplate->MinutiaArr[n].x - lpFeature->MinutiaArr[m].x);
transy = (lpTemplate->MinutiaArr[n].y - lpFeature->MinutiaArr[m].y);
// 将 lpFeature 与 lpTemplate 对齐

```



```

        align(lpFeature, &alignFeature, &lpFeature->MinutiaArr[m],
            rotation, transx, transy);
    // 将两个对齐的指纹特征进行匹配
    alignMatch(&alignFeature, lpTemplate, &globalMatchResult, matchMode);
    // 如果比对结果比最好的比对结果更好, 则更新 alignMax
    if(globalMatchResult.Similarity > alignMax.Similarity)
    {
        alignMax.MMCount = globalMatchResult.MMCount;
        alignMax.Similarity = globalMatchResult.Similarity;
        alignMax.Rotation = rotation;
        alignMax.TransX = transx;
        alignMax.TransY = transy;

        // 如果是快速比对模式, 在相似度达到一定程度后则退出
        if(matchMode==VF_MATCHMODE_IDENTIFY&& alignMax.MMCount >= 8 )
        {
            if(alignMax.Similarity > 100)
            {
                *lpMatchResult = alignMax;
                return;
            }
        }
    }
}

//最终比对结果
*lpMatchResult = alignMax;
}

////////////////////////////////////
//调用各匹配函数
////////////////////////////////////
void patternMatch(FEATUREPTR lpFeature, FEATUREPTR lpTemplate, PMATCHRESULT
lpMatchResult, VF_FLAG matchMode)
{
    vector<int> n_core;
    vector<int> m_core;
    vector<int> n_delta;
    vector<int> m_delta;

    int n, m;
    for(n = 0; n < lpFeature->MinutiaNum; n++) {
        if (lpFeature->MinutiaArr[n].Type == VF_MINUTIA_CORE) {
            n_core.push_back(n);
        }
    }
    for(m = 0; m < lpTemplate->MinutiaNum; m++) {

```

```

        if (lpTemplate->MinutiaArr[m].Type == VF_MINUTIA_CORE) {
            m_core.push_back(m);
        }
    }
    if (n_core.size() > 0 && m_core.size() > 0)
    {
        for(int i = 0; i < n_core.size(); i++)
            for(int j = 0; j < m_core.size(); j++)
            {
                n = (int)n_core[i];
                m = (int)m_core[j];
                coreMatch(lpFeature, lpTemplate, lpMatchResult, matchMode, n, m);
                // 如果是快速比对模式, 在相似度达到一定程度后则退出
                if(matchMode==VF_MATCHMODE_IDENTIFY&& lpMatchResult->MMCount >= 8)
                {
                    if(lpMatchResult->Similarity > 100)
                    {
                        return;
                    }
                }
            }
    }
    if (n_delta.size() > 0 && m_delta.size() > 0)
    {
        deltaMatch(lpFeature, lpTemplate, lpMatchResult, matchMode, n_delta,
            m_delta);
        // 如果是快速比对模式, 在相似度达到一定程度后则退出
        if(matchMode==VF_MATCHMODE_IDENTIFY&&lpMatchResult->MMCount >= 8)
        {
            if(lpMatchResult->Similarity > 100)
            {
                return;
            }
        }
    }
    centralMatch(lpFeature, lpTemplate, lpMatchResult, matchMode);
    if(matchMode == VF_MATCHMODE_IDENTIFY && lpMatchResult->MMCount >= 8)
    {
        if(lpMatchResult->Similarity > 100)
        {
            return;
        }
    }
    globalMatch(lpFeature, lpTemplate, lpMatchResult, matchMode);
}

```

## 10.5 经验分享

本章的程序中用到了动态链接库技术，下面对动态链接库的相关内容做进一步的说明。

1) 动态库的创建与引用。在创建 DLL 时，要根据实际情况选择创建 DLL 的方式，MFC AppWizard 下生成 DLL 文件的方式有三种。一种是常规 DLL 静态链接到 MFC，另一种是常规 DLL 动态链接到 MFC。前者使用的是 MFC 的静态链接库，生成的 DLL 文件长度大，一般使用较少；后者使用 MFC 的动态链接库，生成的 DLL 文件长度小，比较常用。最后一种是 MFC 扩展 DLL，这种 DLL 特点是用来建立 MFC 的派生类，DLL 只被用 MFC 类库所编写的应用程序所调用。

应用程序引用 DLL 采用两种方式：隐式链接和显式链接。隐式链接是在程序开始执行时就将 DLL 文件加载到应用程序中。实现隐式链接时，只要将导入函数关键字 `_declspec(dllimport)` 函数名等写到应用程序相应的头文件中即可。下面的例子通过隐式链接调用 MyDll.dll 库中的 Min 函数。首先生成一个项目 TestDll，在 TestDll.h、TestDll.cpp 文件中分别输入如下代码：

```
//TestDll.h
#pragma comment(lib, "MyDll.lib")
extern "C" _declspec(dllimport) int Max(int a, int b);
extern "C" _declspec(dllimport) int Min(int a, int b);
//TestDll.cpp
#include
#include "TestDll.h"
void main()
{int a;
a=min(2,6);
printf("结果为: %d",a);
}
```

在创建 DllTest.exe 文件之前，要先将 MyDll.dll 和 MyDll.lib 复制到当前工程所在的目录下，也可以复制到 Windows 的 System32 文件夹中。

显式链接是应用程序执行过程中可以随时加载或卸载 DLL 文件的一种链接方式，所以显式链接有很好的灵活性，但是其实现比较复杂。在应用程序中用 LoadLibrary 或 MFC 提供的 AfxLoadLibrary 显式地将自己所做的动态链接库调进来，其文件名即是上述两个函数的参数，此后再用 GetProcAddress() 获取想要引入的函数。在应用程序退出之前，应该用 FreeLibrary 或 MFC 提供的 AfxFreeLibrary 释放动态链接库。

2) 动态库与静态库的区别。在动态库的情况下，有两个文件，一个是引入库 (.lib) 文件，一个是 DLL 文件，引入库文件包含被 DLL 导出的函数的名称和位置，DLL 包含实际的函数和数

据，应用程序使用.lib 文件链接到所需要的 DLL 文件上，库中的函数和数据并不复制到可执行文件中，因此在应用程序的可执行文件中，存放的是 DLL 中所要调用函数的内存地址。当一个或多个应用程序运行时再把程序代码和被调用的函数代码链接起来，可以节省内存资源。因此，DLL 和.lib 文件必须同应用程序一起发布，否则将会产生错误。

在静态库的情况下，函数和数据被编译成一个二进制文件（通常扩展名为\*.lib），编译器在处理代码时将从静态库中恢复这些函数和数据，同应用程序组合在一起编译链接生成可执行文件，这个过程称为“静态链接”。因为应用程序所需的全部内容都是从库中复制出来，所以静态库并不需要与可执行文件一起发布。

## 第 11 章 人脸检测与识别系统

“少小离家老大回，乡音无改鬓毛衰。儿童相见不相识，笑问客从何处来。”一首《回乡偶书》，唱尽诗人贺知章久别家乡告老归来时的感慨。抛开诗中那些久客伤老的哀婉情愫，单看那“儿童相见不相识”一句，若从计算机视觉这一技术角度讲，便涉及人脸识别的问题。人脸识别是指利用分析比较人脸视觉特征信息进行身份鉴别的计算机技术。与其他人体生物特征识别方法（如指纹、虹膜等）相比，人脸图像更容易获取，特别是在非接触环境和不惊动被检测人的情况下，人脸识别的优越性远远超过其他识别技术，因此在安防领域有着非常广泛的应用。本章以一个人脸检测与识别系统为例，介绍人脸的检测和识别原理及编程实现方法。

**本章要点：**

- 人脸检测及定位技术
- 人脸特征提取技术
- 人脸识别技术
- 人脸检测与识别系统功能描述
- 人脸检测与识别系统的总体结构和主要流程
- 人脸检测与识别系统的编程实现

### 11.1 核心技术原理

人脸检测与识别系统的设计主要用到人脸检测及定位技术、人脸特征提取技术和人脸识别技术。下面将详细介绍这些技术。

#### 11.1.1 人脸检测及定位技术

人脸检测及定位技术就是判断图像中是否存在人脸。如果存在，则需要确定人脸的位置和大小等，并将人脸区域从背景图像中分离出来进行标识。下面将具体介绍其原理和方法。

人脸检测的方法主要分为三类：基于先验知识的方法、基于模板匹配的方法和基于统计的方法。基于先验知识是通过抽取灰度、纹理和几何形状等特征形成一定的规则，然后检验这些规则

是否符合人脸先验知识的方法。由于这种方法相对简单,具有计算速度快,耗时小等优点在人脸检测中最为常用。基于模板匹配的方法利用人脸的部分或全部标准特征模板和输入图像中所有的区域进行匹配,通过模板和区域之间的匹配度来检测人脸。此类方法一般速度比较慢,但精度要高一些。基于统计学习的方法依靠统计分析和机器学习来训练人脸样本和非人脸样本,从而将人脸区域检测出来。此方法通常在学习样本充分以及分类器选择得当时,有很高的精度。但这类方法因为要训练尽量多的样本,所以计算量大、耗时多。图 11-1 给出了三类人脸检测方法。

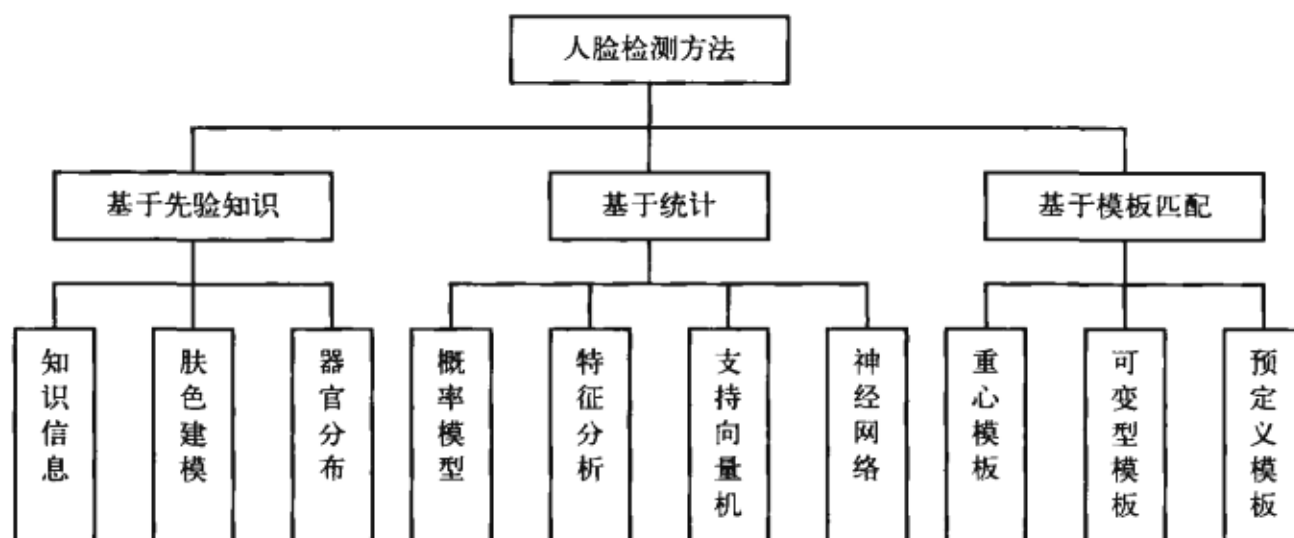


图 11-1 人脸检测方法

对于人脸区域,肤色是占主导地位的色彩。针对彩色图像的人脸检测,肤色检测最为简单有效。基于肤色的人脸检测方法利用图像的彩色信息,在一定的色彩空间中构造肤色模型,该方法在不同的视角中均能检测到人脸,具备稳定性,不受尺寸、表情和人脸姿态变化的影响。本文针对静止的正面人脸彩色图像,主要采用上述人脸检测方法中的肤色建模方法。

人脸检测及定位实现具体步骤如下:

- 1) 选择颜色空间和肤色模型。
- 2) 计算得到相似度灰度图。
- 3) 根据相似度灰度图将图像二值化。
- 4) 垂直方向和水平方向投影。
- 5) 标识人脸区域。

首先要选择合适的颜色空间和肤色模型,下面将具体介绍这两个方面。

人脸的肤色分布具有聚类性,但会受到光照和人种的较大影响。在不同的光照强度下,虽然物体颜色的亮度会产生很大的差异,但是它的色度在很大范围内具有稳定性,基本保持不变。为了减少光照的影响,通常将颜色空间从 RGB 转换到亮度与色度分离的某个颜色空间,比如归一化的 RGB、HSV 或 YCbCr,以去除亮度分量。在色调饱和度平面上,不同人种的肤色变化不大,



肤色的差异更多的是存在于亮度而不是色度。常用的颜色空间及其相互间的转换在本书第 9 章中已经详细介绍。下面介绍几种常见的用于肤色聚类的颜色空间。

### 1. 归一化 RGB 颜色空间

在 RGB 颜色空间中,三个颜色分量  $R$ 、 $G$ 、 $B$  不但表示各自的色彩,也包含了各自的亮度分量。研究表明,如果图像中的两个像素点  $[R, G, B]$  和  $[R', G', B']$  对应成比例且比值不等于 1,如式 (11-1) 所示。则表明两个像素色彩相同,其不同则体现在亮度上。

$$\frac{R}{R'} = \frac{G}{G'} = \frac{B}{B'} \neq 1 \quad (11-1)$$

因此,在色度空间中去除亮度分量,即形成肤色空间,得到归一化的 RGB 颜色空间。如式 (11-2) 所示。

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{bmatrix} \frac{1}{R+G+B} & 0 & 0 \\ 0 & \frac{1}{R+G+B} & 0 \\ 0 & 0 & \frac{1}{R+G+B} \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (11-2)$$

由于  $r+g+b=1$ ,从而忽略任何一个变量,经过上述变换其中两维都是独立的,大大减少了亮度分量的影响,相当于将三维的 RGB 空间降低成二维的  $r-g$  空间。

### 2. HSV 颜色空间

HSV 颜色空间在视觉上是均匀的,与人类的视觉特性有很好的一致性。如果去掉其亮度成分  $V$ ,使用  $H$  和  $S$  分量对图像进行肤色分割,也可以得到很好的效果,但是它也有很大的不足:

1) 三个分量  $H$ 、 $S$ 、 $V$  是由三基色  $R$ 、 $G$ 、 $B$  经过非线性变换得到的,因此计算复杂度高,计算效率较低。

2) HSV 颜色空间中存在着奇异点,即色度点在  $V$  轴上时,其  $S$  值为零,而  $H$  没有定义。而且在奇异点附近  $R$ 、 $G$ 、 $B$  值的较小变化就会引起  $H$ 、 $S$ 、 $V$  值的较大变化。饱和度  $S$  越小,颜色越浅时,色调  $H$  值越不稳定。

### 3. YCbCr 颜色空间

YCbCr 颜色空间的  $Y$  亮度分量和色度分量  $Cb$  与  $Cr$  基本分离,比较适合肤色聚类。除此 YCbCr 颜色空间还显示了其他一些良好的特性:

1) 该空间具有与人类视觉感知过程相类似的构成原理。

2) YCbCr 色彩空间被广泛地应用在电视显示等领域中,也是许多视频压缩编码,如 MPEG 和 JPEG 等标准中普遍采用的颜色表示。

3) YCbCr 的计算过程和空间坐标表示形式简单,与 RGB 之间的转换为线性,容易实现,避

免了非线性空间的奇异性。

4) YCbCr 颜色空间是离散的, 采用 YCbCr 颜色空间易于实现聚类算法。

5) Anil K.Jain 等人绘制了 853571 个肤色点的统计图像, 其结果表明, 肤色在 YCbCr 颜色空间的聚类效果较好。

一般情况下, 彩色图像都是 RGB 颜色空间的, 其他颜色空间都是通过 RGB 转换得到的 (见第 9 章), 而 YCbCr 也是如此。由于是线性转换关系, 其中的亮度分量  $Y$  并不是完全独立于色度信息而存在的, 而 Anil K. Jain 等人的实验也表明, 肤色的聚类区域因为亮度分量的关系而呈现非线性变化的情况, 并且在 YCbCr 颜色空间中, 肤色聚类呈两头尖的椭球形。因此单纯的排除亮度分量  $Y$  的影响, 可能会导致选取的肤色区域不够准确, 降低其鲁棒性。所以, 在肤色检测之前, 要先对图像进行分段线性颜色变换。由 YCbCr 到  $YCb'Cr'$  如式 (11-3) 所示。

$$C'_i(Y) = \begin{cases} C_i(Y) - \overline{C_i(Y)} \cdot \frac{W_{C_i}}{W_{C_i}(Y)} + \overline{C_i(Y)}, & \text{if } (Y < K_i) \text{ or } (K_h < Y) \\ C_i(Y), & \text{if } (Y \in [K_i, K_h]) \end{cases} \quad (11-3)$$

其中,  $i$  表示  $b$  或  $r$ ,  $W_{C_b} = 46.97$ ,  $W_{C_r} = 38.76$ ,  $K_i$  和  $K_h$  为常量分别为 125 和 188,  $C_i(Y)$  为聚类两头尖的椭球形中轴线处的值, 实际上是  $Cb$ 、 $Cr$  两个分量随  $Y$  变化的聚类中心线处的值。

选好颜色空间后, 接下来就是在此颜色空间中进行肤色建模。所谓肤色模型是指用一种代数的、解析的或查找表等形式来表示肤色的聚集特性, 或者表征出某一像素的颜色与肤色的相似程度。常用的肤色模型有直方图模型、椭圆模型和高斯模型。

### 1. 直方图模型

直方图模型是一种非参数化模型。此模型通过选定的颜色空间对肤色进行统计得出其各颜色分量直方图, 由直方图显现出的规律选定阈值, 通过该阈值来对整个图像进行肤色与非肤色的判别。尽管此模型在三维直方图中效果比较好, 但是由于其训练样本数量庞大而且训练时间较长。因此, 在肤色建模中较少使用。

### 2. 椭圆模型

肤色在  $Cb-Cr$  空间中也可以用椭圆分布来描述, 根据式 (11-4) 和式 (11-5) 来匹配椭圆分布两个色度分量的距离。

$$\frac{(x - ec_x)^2}{a^2} + \frac{(y - ec_y)^2}{b^2} = 1 \quad (11-4)$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} C'_b - c_x \\ C'_r - c_y \end{bmatrix} \quad (11-5)$$

其中,  $ec_x$  和  $ec_y$  分别为  $Cb$  和  $Cr$  的统计均值。式中的参数  $ec_x = 1.60$ ,  $ec_y = 2.41$ ,  $a = 25.39$ ,

$b=14.03$ ,  $c_x=114.38$ ,  $c_y=160.02$ ,  $\theta=144.96^\circ$ 。

### 3. 高斯模型

高斯模型主要是利用了统计学的原理。肤色符合正态分布的随机样本,在特征空间中的分布则符合高斯分布,高斯函数平面图如图 11-2 所示。高斯分布的数学表达形式简单且直观,又是统计学中研究比较深入的一种正态模型,因此借助此模型具有一定的优越性。它主要通过统计分析,预测高斯分布的参数,或通过统计直接求得颜色空间中每个分量(一般利用的是该颜色空间中的色度分量)的均值与协方差。这种方法分为两步:首先选择方法确定模型的参数(即均值和协方差),其次利用该模型来判别新的像素或区域是否为肤色。

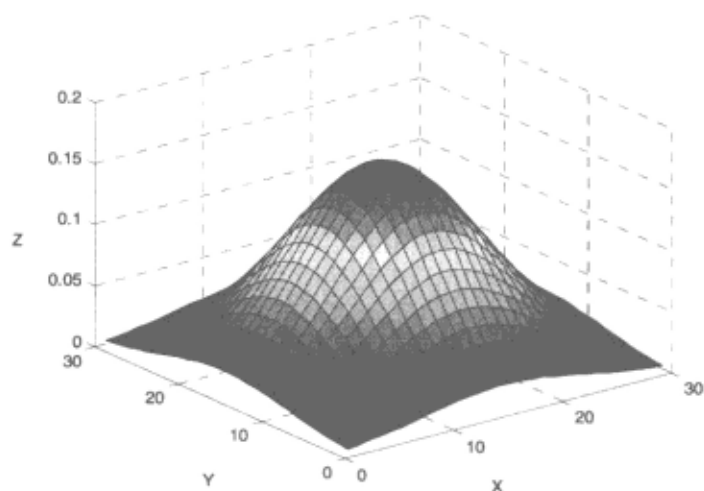


图 11-2 高斯函数平面图

从上述介绍的颜色空间和肤色模型中可以看出,选择在 YCbCr 空间下结合高斯模型对人脸肤色区域进行分割效果最佳,这也是人脸检测的第二步,最终得出相似度灰度图。其具体的分割算法流程如下:

- 1) 读入彩色图像。
- 2) 扫描图像,获取该彩色图像每一像素  $R$ 、 $G$ 、 $B$  颜色分量值。
- 3) 根据公式,将每一像素转换到 YCbCr 空间下,计算  $Cb$ 、 $Cr$  值。
- 4) 根据相似度公式,计算各像素与肤色的相似度值。
- 5) 进行均值滤波,以消除肤色区域中被误判为非肤色的点或是噪声点。
- 6) 得到相似度灰度图,如图 11-3 所示。

在肤色似然度计算过程中,用每一点像素的肤色似然度值除以最大肤色似然度所得到的值,作为该像素点的灰度值,从而得到肤色似然度图像。肤色似然度图像上某点的灰度值表征了这个像素点属于皮肤的概率。而图中具有较高肤色概率值的像素构成的区域就是潜在的人脸区域。其中相似度计算公式如下:



图 11-3 相似度灰度图

$$P(C_b, C_r) = \exp[-0.5(x - m)^T C^{-1}(x - m)] \quad (11-6)$$

$$x = (C_b, C_r)^T, \quad C = E\{(x - m)(x - m)^T\} \quad (11-7)$$

其中,  $x$  为样本像素在 YCbCr 颜色空间的值,  $m = E(x)$  为均值,  $C$  为协方差矩阵。

在第二步得到相似度灰度图之后, 图像中的灰度值还是在  $[0, 255]$  范围内, 不能很好地显示人脸肤色区域, 因此需要选取合适的阈值, 将灰度图像二值化, 最终检测出图像的肤色区域。二值化过程中阈值的选取非常关键。目前有很多阈值选取方法, 如直方图阈值分割法、最大方差阈值分割法和自适应阈值分割法等。本文采用的是基于区域增长算法的自适应阈值分割。

基于区域增长的自适应阈值是根据逐步计算的结果得到的, 其阈值的变化是与第二步的相似度相联系的。逐步减小肤色相似度会得到更多的肤色区域。增长的变化量逐渐减少, 直到已检测到的皮肤区域达到 100%。当相似度小于一定值时, 增长变化将急剧增大, 而不是递减, 这是因为已将人脸肤色区域外围非肤色区域包括进来, 所以使增长最小的相似度或阈值才是最优结果。例如, 相似度以 0.1 的步长从 0.55 减小到 0.05。如果在相似度从 0.35 到 0.25 的改变过程中发生了最小增长的情况, 那么最优的相似度将取为 0.3, 从而根据相似度与阈值的关系得出最优阈值。使用上述方法对肤色相似度灰度图处理就可以得到最终的二值图像, 从而实现了一个完整的肤色分类器。本文中肤色区域用白色表示, 而非肤色区域用黑色表示, 效果如图 11-4 所示。



图 11-4 二值化图

通过人脸检测及定位实现步骤中选择颜色空间和肤色模型、计算相似度和二值化这三步, 根据求得的二值化的值和一些先验知识就能确定人脸区域的上下左右边界, 再由边界形成的外接矩形最终确定人脸区域, 将其标识出来。中间对二值化后的图像进行垂直投影和水平投影, 垂直投影得到水平方向图, 水平投影能够得到垂直方向图, 更能体现出人脸区域的大体范围。

### 11.1.2 人脸特征提取技术

五官是人脸重要的特征点, 五官特征提取是指在图像的给定区域内搜索五官的位置、关键点或轮廓线。五官特征信息既可以验证人脸检测的结果也可以作为人脸识别的依据。本节就人脸五官中的眼睛、嘴巴和鼻子特征进行提取, 下面将详细介绍其原理和方法。

#### 1. 眼睛特征的提取与标定

在提取眼睛特征时, 要同时考虑到颜色和眼睛位置两个特征。首先对检测出来的人脸区域部分进行 canny 边缘检测, 效果如图 11-5 所示。考虑到眉毛与眼睛同属于非肤色部分且距离较近, 对边缘提取后的图像进行水平方向上的投影。因为眉毛正好位于眼睛的上方, 因此水平方向投影眉毛不会影响眼睛部分水平区域的确定。由此通过水平方向的投影能够得出眼睛存在的可能区

域, 计算垂直方向上非肤色点的个数, 根据非肤色点个数的多少就能确定眼睛的水平区域, 其范围为  $A$  和  $B$ 。最后, 在人脸上半部分区域和  $A$ 、 $B$  区域相交区域内分别进行垂直方向投影, 得到的第一个峰值附近的区域为  $C$ 、 $D$ 。然后在  $C$  与  $A$  以及  $D$  与  $B$  确定的两个矩形区域内, 对黑点进行区域膨胀, 可以得到眼睛部分大致轮廓和左右眼角, 最后取黑点的坐标的平均值作为瞳孔的位置, 如图 11-6 所示。



图 11-5 边缘检测效果图

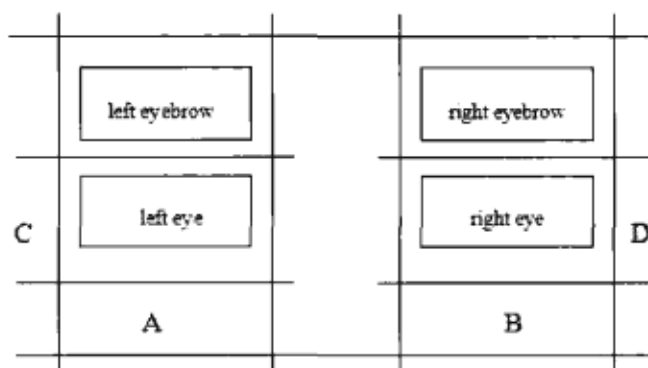


图 11-6 眼睛特征点的提取

## 2. 嘴巴与鼻子特征的提取与标定

眼睛特征确定后, 对于嘴巴和鼻子特征的提取与标定就相对简单, 主要将眼睛作为标准, 根据先验的位置知识和嘴巴与鼻子的颜色特点最终将两者特征记录下来。

### (1) 嘴巴特征提取

对于嘴巴特征点的提取, 考虑了唇色和位置这两个重要的信息。由于在脸部区域内, 唇色与肤色的差别较大。因此, 在检测好的人脸区域内, 如果某一个像素点满足了如下限制条件, 如式 (11-8), 那么这个点极有可能就是嘴唇上的点。

$$\alpha = \cos^{-1} \left( \frac{1/2(2R - GB)}{\sqrt{(R - G)^2 + (R - B)(G - B)}} \right) < 0.2 \quad (11-8)$$

考虑到与眼睛的位置关系, 由先验知识可知, 两眼的瞳孔间距为 1, 嘴中心到两眼中心的距离为 1.0 到 1.3 左右, 在满足这个距离的区域范围内采用类似定位眼睛的方法, 进行区域膨胀, 最后就能确定左右嘴角和嘴中心的位置。

### (2) 鼻子特征提取

对鼻子特征的提取, 仍然是根据与眼睛的位置关系及本身的特性。以两眼瞳孔间距 1 来计算, 从鼻尖到两眼中间的距离为 0.6 到 1。由于鼻孔在人脸中属于非肤色区域, 而且颜色较深, 所以在距离两眼中心以下 0.6 到 1 的距离邻近处搜索颜色较深的区域, 并对该区域进行膨胀处理, 就能得到左右两个鼻孔的位置。而通常人的鼻尖都比较尖, 且在图像中亮度也比较大。所以在两鼻



孔中间位置附近查找亮度最高的点，即为鼻尖所在的位置。

### 11.1.3 人脸识别技术

人脸识别是将静态图像或视频图像中检测出的人脸图像与数据库中的人脸图像进行对比，从中找出与之匹配的人脸的过程，以达到身份识别与鉴定的目的。目前，人脸识别是图像分析与理解的一种最成功的应用，因其在商业、安全和身份认证等方面应用广泛，以及对人脸识别可行性技术的多年研究，使其越来越受重视。

根据人脸表征方式的不同，常用的识别方法可分为以下三种：基于几何特征的识别方法、基于模板匹配的识别方法和基于统计的识别方法。

#### 1. 基于几何特征的识别方法

几何特征法应用于人脸识别中。常用的几何特征有人脸的五官如眼睛、鼻子和嘴巴等局部形状特征、脸型特征以及五官分布的几何特征等。在提取特征时主要通过人脸面部拓扑结构的先验知识。在基于几何特征的识别中，识别总是特征矢量之间的匹配，其中基于欧式距离的识别是最常用的方法。人脸由三庭五眼等特征构成，正因为这些特征在形状、大小或结构上的各种差异才使得人与人不同，因此对这些特征的形状和结构关系的几何描述，可以作为人脸识别的重要特征。当采用几何特征进行正面人脸识别时，一般是通过提取眼睛、嘴巴和鼻子等重要特征点的位置和几何形状作为分类依据。

基于几何特征的识别方法具有存储量小、对光照不敏感、简单且容易理解的优点。但是几何特征的识别方法对图像要求较高，且提取稳定的特征点较为困难，对强烈的表情变化和姿态变化其鲁棒性较差。

#### 2. 基于模板匹配的识别方法

模板匹配法是一种经典的模式识别方法，充分利用了人脸的纹理和灰度特征。识别方法就是将待识别的人脸图像与数据库中所有的模板进行比较，找出最相近的脸。

此方法在匹配时要求两幅图像上的目标要有相同的尺度、取向和光照条件等，所以预先要进行尺度归一化和灰度归一化。最简单的人脸模板是将人脸看成一个椭圆，检测人脸也就是检测图像中的椭圆。另一种方法是将人脸用一组独立的小模板表示，如眼睛、嘴巴和鼻子等。但这些模板的获得必须利用各个特征的轮廓，而传统的基于边缘提取的方法很难获得较高的连续边缘。即使获得了可靠度高的边缘，也很难从中自动提取所需的特征量。后来改用弹性模板方法。弹性模板是由一组根据特征形状的先验知识来设计可调参数的方法。这个参数是由能量函数来决定的，首先利用图像的边缘、峰值、谷值和强度信息以及特征形状的先验知识设计能量函数，然后将参数向能量函数减小的方向调整，当能量函数达到最小时，这组参数所对应的模板形状为最符合的特征形状。在识别率上基于弹性模板的方法比基于几何特征的识别方法要好，但是其速度较慢，



而且要求的内存也大。

### 3. 基于统计的识别方法

基于统计的人脸自动识别方法，包括特征脸方法和隐马尔科夫模型方法。

#### (1) 特征脸方法

特征脸方法是从主成分分析 (PCA) 导出的一种人脸识别和描述技术。PCA 实质上是 K—L 展开的递推实现。K—L 变换是图像压缩技术中的一种最优正交变换，人们将它用于统计特征提取，从而形成子空间法模式识别的基础。若将 K—L 变换用于人脸识别，则需假设人脸处于低维线性空间，由高维图像空间 K—L 变换后，可得到一组新的正交基，由此可以通过保留部分正交基获得正交 K—L 基底。如将子空间对应特征值较大的基底按照图像阵列排列，则可以看出这些正交基呈现出人脸的形状，因此这些正交基也称为特征脸，这种人脸的识别方法也叫特征脸方法。

特征脸方法把人脸图像作为一个整体来编码，而不关心眼、嘴和鼻等单个特征，从而大大降低了识别的复杂度。此方法的主要缺点是目前还没有一个快速的求解特征值和特征向量的算法，每一张新脸入库，都要重新计算特征值和特征向量，费时较多。优点是图像的原始灰度数据可直接用来学习和识别，不需要任何初级或中级处理并且不需要人脸的几何和反射知识。

#### (2) 隐马尔可夫模型

隐马尔可夫模型 (HMM) 是用于描述信号统计特征的一组统计模型。HMM 使用马尔可夫链来模拟信号统计特征的变化，而这种变化是间接地通过观察序列来描述的。因此，隐马尔可夫过程是一个双重的随机过程。一个是马尔可夫链，这是基本随机过程，它描述状态的转移。另一个随机过程描述状态和观测值之间的统计对应关系。在 HMM 中，节点表示状态，有向边表示状态之间的转移，一个状态可以有特征空间中的任意特征，对同一个特征，不同状态表现出这一特征的概率不同。由于 HMM 是一个统计模型，对于同一特征序列，可能会对对应许多状态序列，所以，特征序列与状态序列之间的对应关系是非常正确的。

采用隐马尔可夫模型进行识别的优点是对面部表情和头部姿态的变化不敏感。

本章的人脸检测与识别系统采用了较为简单的几何特征识别方法。因为在特征提取阶段，已经将人脸中的重要特征点眼睛、嘴巴和鼻子的信息记录下来，所以在识别阶段，充分利用记录的特征信息构建关系。根据两只眼睛和嘴巴三个点构成的三角形关系，利用欧式距离分别求出其各个边的长度，指定两个人比较时的误差。在分别打开两个图像并依次完成系统中的操作后，就可以进行匹配，判断是否为同一个人。

## 11.2 系统功能

本章的人脸检测与识别系统首先基于人脸肤色模型对人脸区域进行检测定位，然后利用五官

之间的位置关系进行特征点提取，最后根据特征点来识别人脸与已知人脸是否相同。

### 11.2.1 功能描述

人脸检测与识别系统主要实现以下功能：

- 1) 打开 BMP 位图。
- 2) 对人脸图像进行相似度处理。
- 3) 进行  $Y$ 、 $Cb$ 、 $Cr$  求解。
- 4) 对图像进行二值化。
- 5) 对图像进行滤波去噪。
- 6) 对人脸图像进行水平方向和垂直方向求直方图。
- 7) 用方框标记出人脸区域。
- 8) 对人脸区域进行边缘提取。
- 9) 根据边缘提取结果、人脸先验知识及肤色标记眼睛特征点。
- 10) 根据人脸先验知识与肤色特征标记出嘴巴与鼻子特征点。

### 11.2.2 界面效果

人脸检测与识别系统界面效果如图 11-7 所示。

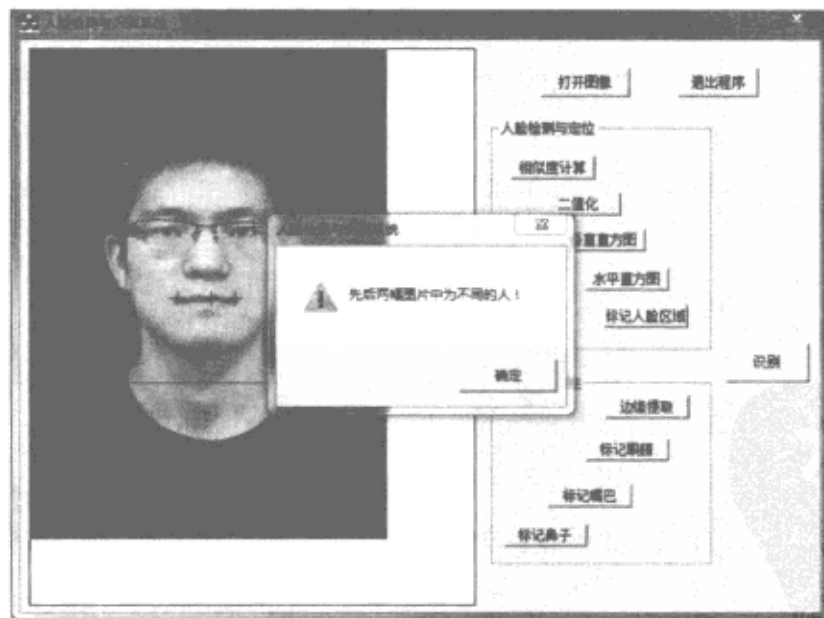


图 11-7 人脸检测与识别系统界面效果图

## 11.3 系统结构与流程

人脸检测与识别系统主要包括人脸检测及定位模块、人脸特征提取模块和人脸识别模块。

### 11.3.1 总体结构

人脸检测与识别系统的总体结构如图 11-8 所示。

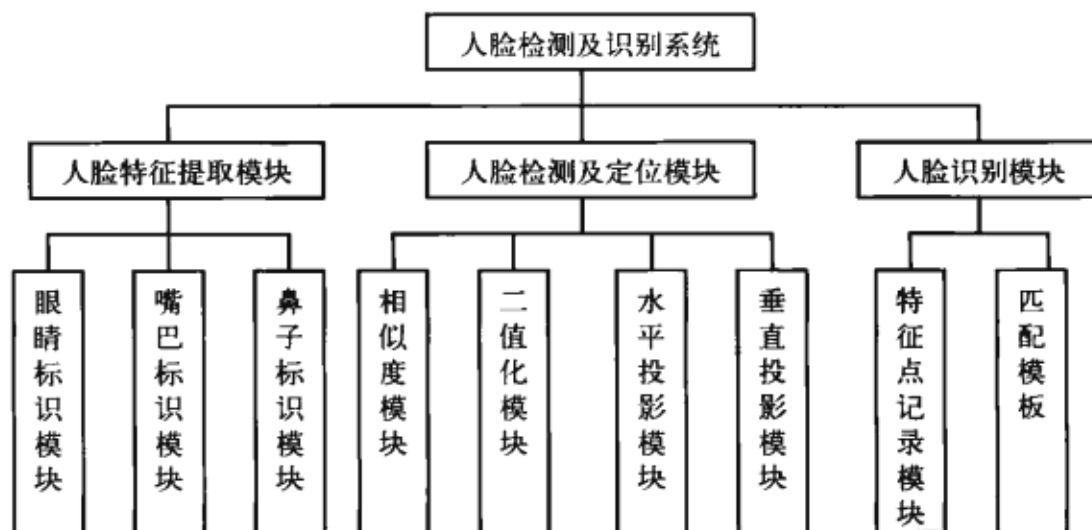


图 11-8 人脸检测与识别系统总体结构

### 11.3.2 主要流程

人脸检测与识别系统的主要流程如图 11-9 所示。

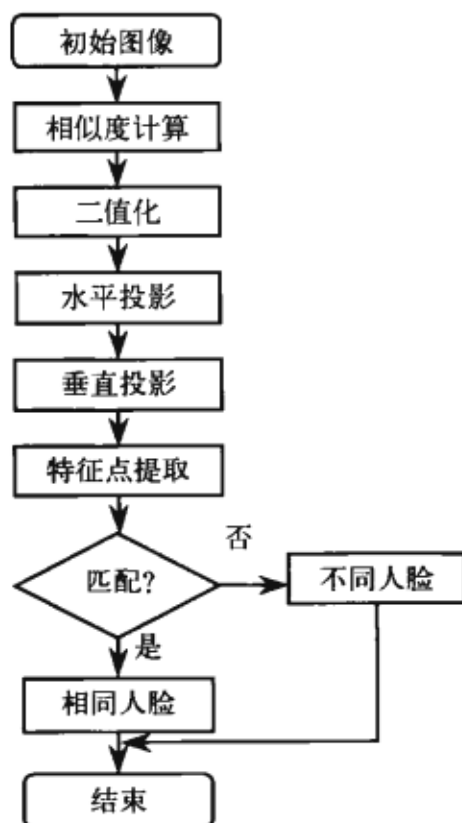


图 11-9 人脸检测与识别系统流程图

## 11.4 编程实现

人脸检测与识别系统采用 VC 2008 开发平台编程实现。

### 11.4.1 人脸检测模块

人脸检测模块主要分为两个小模块：计算相似度和进行二值化。在得出相似度灰度图时，用函数 `int CSimilarHood::CalParameter(CString DirectPath)` 训练好的参数进行计算。将最后的相似度值记录在 `m_pLikeliHoodArray[i][j]` 中。在二值化时，主要是通过相似度进行阈值分割，相似度越大，成为肤色的可能性越大，程序中用参数  $k$  来控制循环计算的次数，当肤色区域增量突然增多时，表明已到了肤色区域的边界，即找到了最佳的阈值。

```

////////////////////////////////////
//计算相似度
////////////////////////////////////
void CSimilarHood::CalLikeHood()
{
    int i,j;
    for(i=0; i<m_nHeight; i++)
    {
        for(j=0; j<m_nWidth; j++)
        {
            double x1,x2;
            TCbCr temp = CalCbCr(m_pSourceData[i][j].rgbRed,m_pSourceData[i][j].
                                rgbGreen,m_pSourceData[i][j].rgbBlue); //计算图像的Cb和Cr分量
            //根据公式和已知变量计算相似度值
            x1 = temp.Cb-bmean;
            x2 = temp.Cr-rmean;
            double t;
            t= x1*(x1*brcov[1][1]-x2*brcov[1][0])+x2*(-x1*brcov[0][1]+x2*brcov[0][0]);
            t /= (brcov[0][0]*brcov[1][1]-brcov[0][1]*brcov[1][0]);
            t /= (-2);
            m_pLikeliHoodArray[i][j] = exp(t); //将计算结果记录在相似度矩阵中
        }
    }

    filter(m_pLikeliHoodArray,m_nWidth,m_nHeight); //均值滤波

    double max = 0.0;
    for(i=0; i<m_nHeight; i++)
        for(j=0; j<m_nWidth; j++)
            if(m_pLikeliHoodArray[i][j] > max)
                max = m_pLikeliHoodArray[i][j]; //求得最大的相似度值
}

```

```

        for(i=0; i<m_nHeight; i++)
        {
            for(j=0; j<m_nWidth; j++)
            {
                //每个相似度值比上最大值得出最终的相似度记录在矩阵中
                m_pLikeliHoodArray[i][j] /= max;
            }
        }
        m_bLikeliHoodReady = true;
    }

    ////////////////////////////////////////
    //计算 Cb 和 Cr
    ////////////////////////////////////////
    TCbCr CSimilarHood::CalCbCr(int R, int G, int B)
    {
        TCbCr res; //存储计算得到的 Cb 和 Cr 的值
        res.Cb =( 128 - 37.797 * R/255 - 74.203 * G/255 + 112 * B/255);
        res.Cr =( 128 + 112 * R/255 - 93.786 * G/255 -18.214 * B/255);
        return res;
    }

    ////////////////////////////////////////
    //滤波
    ////////////////////////////////////////
    void CSimilarHood::filter(double** source,int m_nWidth,int m_nHeight)
    {
        int i,j;
        double **temp;
        //申请一个临时二维数组
        temp = new double*[m_nHeight+2];
        for(i=0;i <=m_nHeight+1; i++)
            temp[i] = new double[m_nWidth+2];

        for(i=0; i<=m_nHeight+1; i++)
        {
            temp[i][0] = 0;
            temp[i][m_nWidth+1] = 0;
        }
        for(j=0; j<=m_nWidth+1; j++)
        {
            temp[0][j] = 0;
            temp[m_nHeight+1][j] = 0;//边界均设为
        }

        //将原数组的值赋予临时数组
        for(i=0; i<m_nHeight; i++)
            for(j=0; j<m_nWidth; j++)

```

```

        temp[i+1][j+1] = source[i][j];
//均值滤波
for(i=0; i<m_nHeight; i++)
{
    for(j=0; j<m_nWidth; j++)
    {
        source[i][j] = 0;
        for(int k=0;k<=2;k++)
        for(int l=0;l<=2;l++)
            source[i][j] += temp[i+k][j+l];
        source[i][j] /= 9;
    }
}
if(temp!=NULL)
{
    for(int i=0;i<=m_nHeight+1;i++)
        if(temp[i]!=NULL) delete temp[i];
    delete temp;
}
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//二值化图像
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
bool CSimilarHood::CalBinary()
{
    if(!m_bLikeliHoodReady)
        return false;
    int i,j;
    BYTE **temp;
    temp = new BYTE*[m_nHeight];
    for(i=0;i <m_nHeight; i++)
    {
        temp[i] = new BYTE[m_nWidth];
        for(j=0; j<m_nWidth; j++)
            temp[i][j] = 0;//初始化一个新矩阵, 用于存放二值化后的结果, 初值都设为 0
    }
    //借助相似度求得进行二值化的阈值
    double min = 1000000000000000000.0;
    int index = -1;
    for(int k=5;k>=0;k--)//用 k 这个变量控制循环的次数
    {
        double sum = 0;
        for(i=0; i<m_nHeight; i++)
        for(j=0; j<m_nWidth; j++)
        {
            if(m_pLikeliHoodArray[i][j]>k*0.1+0.05)//根据相似度来初步判定是否为肤色
                m_pBinaryArray[i][j] = 1;//初步判定为肤色区域
        }
    }
}

```



```

        else
            m_pBinaryArray[i][j] = 0; //初步判定为非人脸区域
            sum += (m_pBinaryArray[i][j]-temp[i][j]);
    }

    if(sum < min)
    {
        min = sum;
        index = 6-k; //借助 index 变量求阈值
    }

    for(i=0; i<m_nHeight; i++)
        for(j=0; j<m_nWidth; j++)
            temp[i][j] = m_pBinaryArray[i][j];
    }

    double optimalThreshold = (7-index)*0.1; //由相似度求得最终二值化的阈值
    for(i=0; i<m_nHeight; i++)
        for(j=0; j<m_nWidth; j++)
        {
            if(m_pLikeliHoodArray[i][j]>optimalThreshold) //相似度与求得的阈值比较
                m_pBinaryArray[i][j] = 1; //最终的人脸区域
            else
                m_pBinaryArray[i][j] = 0; //最终的非人脸区域
        }

    if(temp!=NULL)
    {
        for(int i=0; i<=m_nHeight-1; i++)
            if(temp[i]!=NULL) delete temp[i];
        delete temp;
    }

    m_bBinaryReady = true;
    return true;
}

```

### 11.4.2 人脸定位模块

在得到二值化图后，通过二值化值及先验知识标识人脸。对整个图像分别进行垂直方向投影和水平方向投影，在程序中，根据实际情况选择先进行行运算还是列运算。

```

////////////////////////////////////
// 标记脸部区域，以定位人脸
////////////////////////////////////
void CFaceDetectDlg::OnBtnMarkFace()
{

```

```

if(!method->m_bBinaryReady)
{
    AfxMessageBox("请先计算二值化图!");
    return;
}
m_bShowFace = true;
SetCursor(LoadCursor(NULL, IDC_WAIT));
//初始化变量
int *temp = new int[m_nWndWidth];
int max = 0;
int pos = -1;
for(int j=0; j<m_nWndWidth; j++)
{
    int count = 0;
    for(int i=0; i<m_nWndHeight; i++)
    {
        if(method->m_pBinaryArray[i][j] == 1) count++; //每列中的属于肤色的像素点数
    }
    temp[j] = count;
    if(count > max)
    {
        max = count; //求出各列中最大的肤色像素点数
        pos = j; //含最多肤色像素点的列
    }
}
//求出脸部区域的左边界
int left, right, l, top, bottom;
for(l=pos; l>=0; l--)
{
    if(temp[l]<max*0.2||l==0)
    {
        left = l;
        break;
    }
}
//求出脸部区域的右边界
for(l=pos; l<m_nWndWidth; l++)
{
    if(temp[l]<max*0.3||l==m_nWndWidth-1)
    {
        right = l;
        break;
    }
}
for(int i=0; i<m_nWndHeight; i++)
{
    int count = 0;

```

```

        for(l = left;l<=right;l++)
        {
            //求出每行中从左边界到右边界的属于肤色的像素点数
            if(method->m_pBinaryArray[i][l] == 1) count++;
        }
        if(count>=(right-left)*0.5)
        {
            top = i;//脸部区域的上边界
            break;
        }
    }
    bottom = (int)(top+(right-left)*1.5)>=m_nWndHeight? m_nWndHeight-1:(int)
        (top+(right-left)*1.5);//计算出属于脸部区域的下边界
    CopyBitMap(m_tResPixelArray,m_tOriPixelArray);//恢复为原彩色图像
    //人脸区域的左右边界暂时标识为蓝色
    int i;
    for(i=top;i<=bottom;i++)
    {
        m_tResPixelArray[i][left].rgbBlue=255;
        m_tResPixelArray[i][left].rgbGreen = m_tResPixelArray[i][left].rgbRed = 0;
        m_tResPixelArray[i][right].rgbBlue=255;
        m_tResPixelArray[i][right].rgbGreen = m_tResPixelArray[i][right].rgbRed = 0;
    }
    //人脸区域的上下边界暂时标识为蓝色
    int j;
    for(j=left;j<=right;j++)
    {
        m_tResPixelArray[top][j].rgbBlue=255;
        m_tResPixelArray[top][j].rgbGreen = m_tResPixelArray[top][j].rgbRed = 0;
        m_tResPixelArray[bottom][j].rgbBlue=255;
        m_tResPixelArray[bottom][j].rgbGreen = m_tResPixelArray[bottom][j].
            rgbRed = 0;
    }
    MakeBitMap();
    SetCursor(LoadCursor(NULL, IDC_ARROW));
    if(!m_bFaceOK)m_bFaceOK = true;

    CopyBitMap(m_tResPixelArray,m_tOriPixelArray);//恢复为原彩色图像
    CRect rect(left,top,right,bottom);//用矩形框对脸部区域进行标记
    m_rFaceRegion = rect;
    MakeBitMap();
}
////////////////////////////////////
// 水平方向直方图
////////////////////////////////////
void CFaceDetectDlg::OnBtnHistogramH()
{

```

```

        if(!method->m_bBinaryReady)
        {
            AfxMessageBox("请先计算二值图");
            return;
        }
        m_bShowFace = false;
        SetCursor(LoadCursor(NULL, IDC_WAIT));
        //按列来看, 绘制水平方向直方图
        for(int j=0; j<m_nWndWidth; j++)
        {
            int count = 0;
            for(int i=0; i<m_nWndHeight; i++)
            {
                if(method->m_pBinaryArray[i][j] == 1) count++; //属于肤色的像素点的个数
                m_tResPixelArray[i][j].rgbBlue = m_tResPixelArray[i][j].rgbGreen =
                m_tResPixelArray[i][j].rgbRed = 255; //不属于肤色的像素点用白色表示
            }
            int i;
            for(i=m_nWndHeight-1; i>=m_nWndHeight-count; i--)
            {
                m_tResPixelArray[i][j].rgbBlue = m_tResPixelArray[i][j].rgbGreen =
                m_tResPixelArray[i][j].rgbRed = 0; //属于肤色区域的像素点用黑色表示
            }
        }
        MakeBitMap();
        SetCursor(LoadCursor(NULL, IDC_ARROW));
    }
    ////////////////////////////////////////
    // 垂直方向的直方图
    ////////////////////////////////////////
    void CFaceDetectDlg::OnBtnHistogramV()
    {
        if(!method->m_bBinaryReady)
        {
            AfxMessageBox("请先计算二值图");
            return;
        }
        m_bShowFace = false;
        SetCursor(LoadCursor(NULL, IDC_WAIT));
        //按行来看, 绘制垂直直方图
        for(int i=0; i<m_nWndHeight; i++)
        {
            int count = 0;
            for(int j=0; j<m_nWndWidth; j++)
            {
                if(method->m_pBinaryArray[i][j] == 1) count++; //每行中属于肤色的像素点数
                m_tResPixelArray[i][j].rgbBlue = m_tResPixelArray[i][j].rgbGreen =

```



```

        m_tResPixelArray[i][j].rgbRed = 255;//不属于肤色的用白色表示
    }
    int j;
    for(j=0; j<count; j++)
    {
        m_tResPixelArray[i][j].rgbBlue = m_tResPixelArray[i][j].rgbGreen =
        m_tResPixelArray[i][j].rgbRed = 0;//属于肤色的用黑色表示
    }
}
MakeBitMap();
SetCursor(LoadCursor(NULL, IDC_ARROW));

```

### 11.4.3 人脸特征点提取模块

人脸特征点提取模块主要是对眼睛、嘴巴和鼻子的特征点进行提取并标注。

#### 1. 眼睛的标注

对于眼睛的标注以 FaceRegion.top+6\*slidwinHeight 为矩形顶，MidFaceV-slidwinHeight 为底，FaceRegion.left 为矩形左边，FaceRegion.right 为矩形右边。

```

////////////////////////////////////
// 标记眼睛区域
////////////////////////////////////
void CFaceDetectDlg::OnBtnMarkEye()
{
    int i,j;
    if(!m_bFaceOK)
    {
        AfxMessageBox("请先确定脸部区域");
        return;
    }
    //左右眼的水平区域
    CPoint LeftEyeAreaH(-1,-1),RightEyeAreaH(-1,-1);
    CPoint LeftEyeAreaV(-1,-1),RightEyeAreaV(-1,-1);
    int nLeft,nRight,nTop,nBottom;
    nLeft = m_rFaceRegion.left-5 > 0 ? m_rFaceRegion.left-5:0;
    nRight=m_rFaceRegion.right+5<m_nWndWidth? m_rFaceRegion.right+5:m_nWndWidth-1;
    nTop= m_rFaceRegion.top-5 > 0 ?
    m_rFaceRegion.top-5:0;
    nBottom=m_rFaceRegion.bottom+5< m_nWndHeight?m_rFaceRegion.bottom+5:m_nWnd
        Height-1;
    //边缘检查
    DoLOG(nLeft,nRight,nTop,nBottom,m_tOriPixelArray,m_tResPixelArray);
    //////////////////////////////////
    //确认两只眼睛的水平区域
    //////////////////////////////////
    int nSlidWinWidth = (m_rFaceRegion.right - m_rFaceRegion.left)/6/2;

```

```

int nSlidWinHeight = (m_rFaceRegion.bottom - m_rFaceRegion.top)/15/2;
int nMidFaceH = (m_rFaceRegion.right+m_rFaceRegion.left)/2;
//人脸区域的水平中间位置
int nMidFaceV = (m_rFaceRegion.bottom+m_rFaceRegion.top)/2;
//人脸区域的竖直中间位置

int *tempArray = new int[m_nWndWidth];
for(i = 0; i<m_nWndWidth; i++) tempArray[i] = 0;

for(i=nMidFaceV-nSlidWinHeight; i > m_rFaceRegion.top+6*nSlidWinHeight; i--)
for(j=m_rFaceRegion.left+nSlidWinWidth; j<m_rFaceRegion.right-nSlidWinWidth; j++)
{
    int count = 0;
    for(int p= -nSlidWinHeight ;p<nSlidWinHeight;p++)
    for(int q= -nSlidWinWidth ;q<nSlidWinWidth;q++)
    {
        if(m_tResPixelArray[i+p][j+q].rgbRed == 0)    count++;
    }
    if(count >= nSlidWinWidth*nSlidWinHeight/3)
    {
        m_tResPixelArray[i][j].rgbRed = 255;
        tempArray[j] ++;
    }
}

MakeBitMap();
AfxMessageBox("眼睛的区域鉴别");

CList<CPoint,CPoint> myList1(sizeof(CPoint));
CList<CPoint,CPoint> myList2(sizeof(CPoint));
int flag = 0;
CPoint tPoint(-1,-1);
for(i = 0; i<m_nWndWidth; i++)
{
    if(tempArray[i] > 0 && flag ==0)
    {
        tPoint.x = i;
        flag = 1;
    }
    if(tempArray[i] == 0 && flag ==1)
    {
        tPoint.y = i;
        myList1.AddTail(tPoint);
        flag = 0;
    }
}
}

```



```

delete tempArray;
//去掉长度太小的候选者
for(i=0; i<myList1.GetCount();i++)
{
    CPoint temp(-1,-1);
    temp = myList1.GetAt(myList1.FindIndex(i));
    int minVal = (m_rFaceRegion.right - m_rFaceRegion.left)/20;
    if((temp.y-temp.x)>=minVal)
        myList2.AddTail(temp);
}
myList1.RemoveAll();
//合并相邻很近的区域
bool quit = 1;
while(quit)
{
    bool doJoin = false;
    for(int i=0; i<myList2.GetCount()-1;i++)
    {
        CPoint temp1(-1,-1),temp2(-1,-1);
        temp1 = myList2.GetAt(myList2.FindIndex(i));
        temp2 = myList2.GetAt(myList2.FindIndex(i+1));
        if((temp2.x-temp1.y)<=(m_rFaceRegion.right - m_rFaceRegion.left)/40)
        {
            temp1.y = temp2.y;
            myList2.RemoveAt(myList2.FindIndex(i));
            myList2.RemoveAt(myList2.FindIndex(i));
            if(i == 0) myList2.AddHead(temp1);
            else myList2.InsertAfter(myList2.FindIndex(i-1),temp1);
            doJoin = true;
            break;
        }
    }
    if(!doJoin)quit = 0;
}

//没有找到眼睛区域
if(myList2.GetCount()<2)
{
    CPoint t=myList2.GetHead();
    if((t.y-t.x)>(m_rFaceRegion.right - m_rFaceRegion.left)/2)
    {
        LeftEyeAreaH.x = t.x;
        LeftEyeAreaH.y = t.x+(t.y-t.x)/3;
        RightEyeAreaH.x = t.y-(t.y-t.x)/3;
        RightEyeAreaH.y = t.y;
    }
    else

```

```

    {
        AfxMessageBox("确认眼睛位置失败, 请手动标定");
        return;
    }
}
//仅有两个区域
else if(myList2.GetCount()==2)
{
    LeftEyeAreaH = myList2.GetHead();
    RightEyeAreaH = myList2.GetTail();
}
else //多于两个区域
{
    int ldis = -100000;
    int rdis = 100000;
    for(i=0; i<myList2.GetCount();i++)
    {
        CPoint temp(-1,-1);
        temp = myList2.GetAt(myList2.FindIndex(i));
        //确认右眼
        if((temp.x+temp.y)/2 > nMidFaceH)
        {
            if(((temp.x+temp.y)/2-nMidFaceH)<rdis)
            {
                rdis = (temp.x+temp.y)/2-nMidFaceH;
                RightEyeAreaH = temp;
            }
        }
        //确认左眼
        else
        {
            if(((temp.x+temp.y)/2-nMidFaceH)>ldis)
            {
                ldis = (temp.x+temp.y)/2-nMidFaceH;
                LeftEyeAreaH = temp;
            }
        }
    }
}
myList2.RemoveAll();
//////////////////////////
//确认两只眼睛的垂直区域
//////////////////////////
//左眼
if(LeftEyeAreaH != CPoint(-1,-1))
{
    int *tArray = new int[m_nWndHeight];

```

```

int i,j;
for(i = 0; i<m_nWndHeight; i++) tArray[i] = 0;
for(i=nMidFaceV-nSlidWinHeight; i>m_rFaceRegion.top+6*nSlidWinHeight; i--)
for(j=LeftEyeAreaH.x; j<=LeftEyeAreaH.y;j++)
if(m_tResPixelArray[i][j].rgbRed == 255 && m_tResPixelArray[i][j].
    rgbGreen == 0)
    tArray[i] ++;
CList<CPoint,CPoint&> myListA(sizeof(CPoint));
CList<CPoint,CPoint&> myListB(sizeof(CPoint));
int flag = 0;
CPoint tPoint(-1,-1);
for(i = nMidFaceV-nSlidWinHeight; i>m_rFaceRegion.top+6*nSlidWinHeight; i--)
{
    if(tArray[i] > 0 && flag ==0)
    {
        tPoint.x = i;
        flag = 1;
    }
    if(tArray[i] == 0 && flag ==1)
    {
        tPoint.y = i;
        myListA.AddTail(tPoint);
        flag = 0;
    }
}
delete tArray;
//去掉长度太小的候选者
for(i=0; i<myListA.GetCount();i++)
{
    CPoint temp(-1,-1);
    temp = myListA.GetAt(myListA.FindIndex(i));
    int minVal = (m_rFaceRegion.bottom - m_rFaceRegion.top)/100;
    if((temp.x-temp.y)>=minVal)
        myListB.AddTail(temp);
}
myListA.RemoveAll();
//合并相邻很近的区域
bool quit = 1;
while(quit)
{
    bool doJoin = false;
    for(int i=0; i<myListB.GetCount()-1;i++)
    {
        CPoint temp1(-1,-1),temp2(-1,-1);
        temp1 = myListB.GetAt(myListB.FindIndex(i));
        temp2 = myListB.GetAt(myListB.FindIndex(i+1));
        if((temp1.y-temp2.x)<=(m_rFaceRegion.bottom - m_rFace

```

```

        Region.top)/100)
    {
        templ.y = temp2.y;
        myListB.RemoveAt(myListB.FindIndex(i));
        myListB.RemoveAt(myListB.FindIndex(i));
        if(i == 0) myListB.AddHead(templ);
        else myListB.InsertAfter(myListB.FindIndex(i-1), templ);
        doJoin = true;
        break;
    }
}
if(!doJoin) quit = 0;
}
if(myListB.GetCount()==0)
{
    AfxMessageBox("无法确定左眼的位置");
}
else
{
    LeftEyeAreaV = myListB.GetHead();

    double sumX = 0.0;
    double sumY = 0.0;
    int sum = 0;
    m_LeftEyeLeftCorner.x = 100000;
    m_LeftEyeRightCorner.x = -1;

    for(i=LeftEyeAreaV.x; i>= LeftEyeAreaV.y;i--)
    for(j=LeftEyeAreaH.x; j<=LeftEyeAreaH.y;j++)
    if(m_tResPixelArray[i][j].rgbGreen == 0)
    {
        if(j<m_LeftEyeLeftCorner.x) //初步判断左眼角位置
        {
            m_LeftEyeLeftCorner.x = j;
            m_LeftEyeLeftCorner.y = i;
        }
        if(j>m_LeftEyeRightCorner.x)//初步判定右眼角位置
        {
            m_LeftEyeRightCorner.x = j;
            m_LeftEyeRightCorner.y = i;
        }
        sumX += j;
        sumY += i;
        sum++;
    }
    //确定左眼的坐标
    m_LeftEye.x = (int)(sumX/sum);

```



```

        m_LeftEye.y = (int)(sumY/sum);

        m_bLeftEyeOK = TRUE;
        m_bLeftEyeLeftCornerOK = TRUE;
        m_bLeftEyeRightCornerOK = TRUE;
    }
    myListB.RemoveAll();
}
//右眼
if(RightEyeAreaH != CPoint(-1,-1))
{
    int *tArray = new int[m_nWndHeight];
    int i,j;
    for(i = 0; i<m_nWndHeight; i++) tArray[i] = 0;
    for(i=nMidFaceV-nSlidWinHeight; i>m_rFaceRegion.top+6*nSlidWinHeight; i--)
    for(j=RightEyeAreaH.x; j<=RightEyeAreaH.y;j++)
    if(m_tResPixelArray[i][j].rgbRed == 255 && m_tResPixelArray[i][j].
        rgbGreen == 0)
        tArray[i] ++;
    CList<CPoint,CPoint&> myListA(sizeof(CPoint));
    CList<CPoint,CPoint&> myListB(sizeof(CPoint));
    int flag = 0;
    CPoint tPoint(-1,-1);
    for(i = nMidFaceV-nSlidWinHeight; i>m_rFaceRegion.top+6*nSlidWinHeight; i--)
    {
        if(tArray[i] > 0 && flag ==0)
        {
            tPoint.x = i;
            flag = 1;
        }
        if(tArray[i] == 0 && flag ==1)
        {
            tPoint.y = i;
            myListA.AddTail(tPoint);
            flag = 0;
        }
    }
    delete tArray;
    //去掉长度太小的候选者
    for(i=0; i<myListA.GetCount();i++)
    {
        CPoint temp(-1,-1);
        temp = myListA.GetAt(myListA.FindIndex(i));
        int minVal = (m_rFaceRegion.bottom - m_rFaceRegion.top)/100;
        if((temp.x-temp.y)>=minVal)
            myListB.AddTail(temp);
    }
}

```

```

myListA.RemoveAll();
//合并相邻很近的区域
bool quit = 1;
while(quit)
{
    bool doJoin = false;
    for(int i=0; i<myListB.GetCount()-1;i++)
    {
        CPoint temp1(-1,-1),temp2(-1,-1);
        temp1 = myListB.GetAt(myListB.FindIndex(i));
        temp2 = myListB.GetAt(myListB.FindIndex(i+1));
        if((temp1.y-temp2.x)<=(m_rFaceRegion.bottom-
m_rFaceRegion.top)/50)
        {
            temp1.y = temp2.y;
            myListB.RemoveAt(myListB.FindIndex(i));
            myListB.RemoveAt(myListB.FindIndex(i));
            if(i == 0)myListB.AddHead(temp1);
            elsemyListB.InsertAfter(myListB.FindIndex(i-1),temp1);
            doJoin = true;
            break;
        }
    }
    if(!doJoin)quit = 0;
}
if(myListB.GetCount()==0)
{
    AfxMessageBox("无法确定右眼的位置");
}
else
{
    if(myListB.GetCount()==1)
        RightEyeAreaV = myListB.GetHead();
    else
    {
        if(myListB.GetCount()==1)
            RightEyeAreaV = myListB.GetHead();
        else
        {
            CPoint tt = myListB.GetHead();
            int index = myListB.GetCount();
            while(tt.y > LeftEyeAreaV.x && index > 0)
            {
                index --;
                tt = myListB.GetAt(myListB.FindIndex(myListB.GetCount()-
index));
            }
            RightEyeAreaV = tt;
        }
    }
    double sumX = 0.0;
    double sumY = 0.0;
}

```



```

int sum = 0;
m_RightEyeLeftCorner.x = 100000;
m_RightEyeRightCorner.x = -1;
for(i=RightEyeAreaV.x; i>=RightEyeAreaV.y;i--)
for(j=RightEyeAreaH.x; j<=RightEyeAreaH.y;j++)
if(m_tResPixelArray[i][j].rgbGreen == 0)
{
    if(j<m_RightEyeLeftCorner.x)
    {
        //初步确定右眼角位置
        m_RightEyeLeftCorner.x = j;
        m_RightEyeLeftCorner.y = i;
    }
    if(j>m_RightEyeRightCorner.x)
    {
        m_RightEyeRightCorner.x = j;
        m_RightEyeRightCorner.y = i;
    }
    sumX += j;
    sumY += i;
    sum++;
}
//确定右眼位置
m_RightEye.x = (int)(sumX/sum);
m_RightEye.y = (int)(sumY/sum);
m_bRightEyeOK = TRUE;
m_bRightEyeLeftCornerOK = TRUE;
m_bRightEyeRightCornerOK = TRUE;
}
myListB.RemoveAll();
}
CopyBitMap(m_tResPixelArray,m_tOriPixelArray);
MakeBitMap();
}

```

## 2. 嘴的标注

对于嘴部的标注,首先是通过嘴本身肤色的特性找到其大体位置并将其显示为红色,由先验知识在人脸左眼及右眼与上下嘴唇之间形成的小矩形区域内,根据嘴部各位置红色像素点数的不同最终确定嘴中间及左右嘴角。

```

////////////////////////////////////
// 标记嘴
////////////////////////////////////
void CFaceDetectDlg::OnBtnMarkMouse()
{
    int i,j;

```

```

if(!(m_bLeftEyeOK&& m_bRightEyeOK))
{
    AfxMessageBox("请先确定眼睛");
    return;
}
//左右眼的水平区域
int nLeft,nRight,nTop,nBottom;

nLeft = m_rFaceRegion.left-5 > 0 ? m_rFaceRegion.left-5:0;
nRight=m_rFaceRegion.right+5<m_nWndWidth? m_rFaceRegion.right+5:m_nWndWidth-1;
nTop= m_rFaceRegion.top-5 > 0 ? m_rFaceRegion.top-5:0;
nBottom=m_rFaceRegion.bottom+5< m_nWndHeight?m_rFaceRegion.bottom+5:m_
    nWndHeight-1;
SetPixelArray(m_tResPixelArray,0);//初始为
for(i=nTop; i<=nBottom; i++)
for(j=nLeft; j<=nRight; j++)
{
    BYTE R,G,B;
    double temp,delta;
    //获取原始图像的 RGB 颜色分量
    R = m_tOriPixelArray[i][j].rgbRed;
    G = m_tOriPixelArray[i][j].rgbGreen;
    B = m_tOriPixelArray[i][j].rgbBlue;
    if((R==G) && (G==B)) temp = 0;
    //计算嘴部肤色的像素点值
    else temp=(double)(0.5*(2*R-G-B)/sqrt((double)(R-G)*(R-G)+(double)
        (R-B)*(G-B)));
    delta = acos(temp);
    if(delta < 0.2)//由先验知识得出判断
    {
        m_tResPixelArray[i][j].rgbRed = 255; //暂时将嘴部改为红色显示
    }
    else m_tResPixelArray[i][j].rgbRed = 0;
}
MakeBitMap();
AfxMessageBox("嘴的肤色鉴定");
//双目斜角
double tanThta;
if(m_RightEye.y == m_LeftEye.y) tanThta = 0;
else tanThta = (m_RightEye.y - m_LeftEye.y)/(m_RightEye.x - m_LeftEye.x);
//双目距离
int EyesDis = (m_RightEye.x-m_LeftEye.x)*(m_RightEye.x-m_LeftEye.x);
EyesDis += (m_RightEye.y-m_LeftEye.y)*(m_RightEye.y-m_LeftEye.y);
EyesDis = (int)sqrt((float)EyesDis);
//双目平均高度
int EyeV = (m_RightEye.y + m_LeftEye.y)/2;
//可能的嘴的区域

```

```

int MouthUp = (EyeV+1.0*EyesDis) > nBottom ? nBottom: (int) (EyeV+1.0*EyesDis);
int MouthDown = (EyeV+1.5*EyesDis) > nBottom ? nBottom: (int) (EyeV+1.5*EyesDis);

//确定左右嘴角及嘴中间的位置
int* Y_Array = new int[MouthDown-MouthUp];
for(i =0 ;i < MouthDown-MouthUp ;i++) Y_Array[i] = 0;//初始行数并置值
int* X_Array = new int[EyesDis];//初始列数
for(i =0 ;i < EyesDis ;i++) X_Array[i] = 0;
//从嘴部最上位置到最下, 从左眼到右眼形成的矩形区域内搜索
for(i = MouthUp ; i < MouthDown; i++)
for(j = m_LeftEye.x; j< m_RightEye.x; j++)
{
    if(m_tResPixelArray[i][j].rgbRed == 255)//显示红色的部分
    {
        //小矩形区域内标示嘴部像素点
        Y_Array[i-MouthUp] ++;
        X_Array[j-m_LeftEye.x] ++;
    }
}
int maxY = 0;
for(i =0 ;i < MouthDown-MouthUp ;i++)
{
    if(Y_Array[i]>maxY)
    {
        maxY = Y_Array[i];//求出含有最多唇色像素点的行
        //得到嘴部中间位置的纵坐标
        m_MidMouth.y = i+MouthUp - (MouthDown-MouthUp)/10;
    }
}
//左右嘴角的纵坐标
m_LeftMouthCorner.y =(int) (m_MidMouth.y - tanThta*EyesDis/2);
m_RightMouthCorner.y =(int) (m_MidMouth.y + tanThta*EyesDis/2);
for(i =0 ;i < EyesDis ;i++)
{
    if(X_Array[i]>0)
    {
        m_LeftMouthCorner.x = i+m_LeftEye.x;//左唇角的横坐标位置
        break;
    }
}
for(i = EyesDis -1; i >=0 ;i--)
{
    if(X_Array[i]>0)
    {
        m_RightMouthCorner.x = m_LeftEye.x+i;//右唇角的横坐标位置
        break;
    }
}

```

### 3. 鼻子的标注

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// 标记鼻子
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void CFaceDetectDlg::OnBtnMarkNose()
{
    int i,j;
    if(!(m_bLeftEyeOK&& m_bRightEyeOK))
    {
        AfxMessageBox("请先确定眼睛");
        return;
    }
    //左右眼的水平区域
    int nLeft,nRight,nTop,nBottom;
    nLeft    = m_rFaceRegion.left-5 > 0 ? m_rFaceRegion.left-5:0;
    nRight=m_rFaceRegion.right+5<m_nWndWidth? m_rFaceRegion.right+5:m_nWndWidth-1;
    nTop= m_rFaceRegion.top-5 > 0 ? m_rFaceRegion.top-5:0;
    nBottom=m_rFaceRegion.bottom+5< m_nWndHeight?m_rFaceRegion.bottom+5:m_
        nWndHeight-1;
    SetPixelArray(m_tResPixelArray,0);
    for(i=nTop; i<=nBottom; i++)
    for(j=nLeft; j<=nRight; j++)
    {

```

```

double Y;
Y = 0.30*m_tOriPixelFormatArray[i][j].rgbRed+0.59*m_tOriPixelFormatArray[i][j].rgbGreen
    +0.11*m_tOriPixelFormatArray[i][j].rgbBlue;//将彩色图转为灰度图
if(Y<100)//阈值分割得到鼻孔较黑部分
{
    m_tResPixelFormatArray[i][j].rgbRed = 255; //符合判断的暂时将鼻子显示为红色
}
else m_tResPixelFormatArray[i][j].rgbRed = 0;
}
MakeBitMap();
AfxMessageBox("鼻子的肤色鉴定");
//双目斜角
double tanThta;
if(m_RightEye.y == m_LeftEye.y) tanThta = 0;
else tanThta = (m_RightEye.y - m_LeftEye.y)/(m_RightEye.x - m_LeftEye.x);
//双目距离
int EyesDis = (m_RightEye.x-m_LeftEye.x)*(m_RightEye.x-m_LeftEye.x);
EyesDis += (m_RightEye.y-m_LeftEye.y)*(m_RightEye.y-m_LeftEye.y);
EyesDis = (int)sqrt((float)EyesDis);
//双目平均高度
int EyeV = (m_RightEye.y + m_LeftEye.y)/2;
//可能的鼻子的区域
int NoseUp = (EyeV+0.5*EyesDis) > nBottom ? nBottom:(int) (EyeV+0.5*EyesDis);
int NoseDown = (EyeV+0.8*EyesDis) > nBottom ? nBottom:(int) (EyeV+0.8*EyesDis);
int* Y_Array = new int[NoseDown-NoseUp];
for(i =0 ;i < NoseDown-NoseUp ;i++) Y_Array[i] = 0;
//记录小矩形区域的行数并置其中值为 0
int* X_Array = new int[EyesDis];
for(i =0 ;i < EyesDis ;i++) X_Array[i] = 0;
for(i = NoseUp ; i < NoseDown; i++)
for(j = m_LeftEye.x+EyesDis/5; j< m_RightEye.x-EyesDis/5; j++)
//由先验知识得到三庭五眼
{
    if(m_tResPixelFormatArray[i][j].rgbRed == 255)
    {
        //小矩形区域内标示鼻子像素点
        Y_Array[i-NoseUp] ++;
        X_Array[j-m_LeftEye.x] ++;
    }
}
int maxY = 0;
for(i =0 ;i < NoseDown-NoseUp ;i++)
{
    if(Y_Array[i]>maxY)
    {
        maxY = Y_Array[i];//求出含有最多鼻孔颜色像素点的行
        m_MidNose.y = i+NoseUp;//确定鼻子中间纵坐标
    }
}

```

```

    }
}
m_LeftNostril.y =(int)(m_MidNose.y - tanThta*EyesDis/2);
m_RightNostril.y =(int)(m_MidNose.y + tanThta*EyesDis/2);
for(i =0 ;i < EyesDis ;i++)
{
    if(X_Array[i]>0)
    {
        m_LeftNostril.x = i+m_LeftEye.x;//确定左鼻孔的纵坐标
        break;
    }
}
for(i = EyesDis-1; i >=0 ;i--)
{
    if(X_Array[i]>0)
    {
        m_RightNostril.x = i+m_LeftEye.x;//确定右鼻孔的纵坐标
        break;
    }
}
//唇中点较薄
int min = 1000000;
for(i = (int)(EyesDis/3+0.5) ; i <= (int)(2*EyesDis/3+0.5);i++)
{
    if(X_Array[i]<min)
    {
        min = X_Array[i];//含最少红色标记的部分即为鼻子中间部分
        m_MidNose.x = m_LeftEye.x+i;
    }
}
m_MidNose.x = (m_MidNose.x+(m_LeftEye.x+EyesDis/2))/2;//确定鼻子的横坐标
////////////////////////////////////
// 求取特征点并进行保存
////////////////////////////////////
EM = (float)(m_MidMouth.x -m_LeftEye.x)*(float)(m_MidMouth.x -m_LeftEye.x);
EM += (float)(m_MidMouth.y-m_LeftEye.y)*(float)(m_MidMouth.y-m_LeftEye.y);
EM = sqrt((float)EM);//左眼与嘴部的欧氏距离
if(em)
    eml=EM;
else
    em=EM;
Feature1[0]=ed;
Feature1[1]=me;
Feature1[2]=em;

ME = (float)(m_RightEye.x -m_MidMouth.x)*(float)(m_RightEye.x -m_MidMouth.x);
ME+=(float)(m_LeftEye.y-m_MidMouth.y)*(float)(m_LeftEye.y-m_MidMouth.y);

```



```

        ME= sqrt((float)ME); //右眼和嘴部的欧氏距离
    if(me)
        mel=ME;
    else
        me=ME;
    Feature2[0]=ed1;
    Feature2[1]= mel;
    Feature2[2]= em1;

    m_bMidNoseOK = TRUE;
    m_bLeftNostrilOK = TRUE;
    m_bRightNostrilOK = TRUE;
    //判断是否打开了两幅图
    if( ! Feature2[2])
        m_bMatchOK=false;
    else
        m_bMatchOK=TRUE;

    CopyBitMap(m_tResPixelArray,m_tOriPixelArray);
    MakeBitMap();
}

```

#### 11.4.4 人脸匹配模块

人脸匹配模块主要实现人脸特征比对。通过两只眼睛和嘴巴形成的三角区域的三边长度进行比较（其误差锁定在 0.1 范围内）来判断先后打开的两幅图像是否为同一个人。

```

////////////////////////////////////
// 记录特征数据
////////////////////////////////////
    float Feature1[3]; //记录第一次打开图像脸部三角区域三边长
    float Feature2[3]; //记录第二次打开图像脸部三角区域三边长
    float EM,em1,em,ME,mel,me,ed1,ed; //中间变量
    //求取特征点并进行保存
    EM = (float)(m_MidMouth.x -m_LeftEye.x)*(float)(m_MidMouth.x -m_LeftEye.x);
    EM += (float)(m_MidMouth.y-m_LeftEye.y)*(float)(m_MidMouth.y-m_LeftEye.y);
    EM = sqrt((float)EM);
    if(em)
        em1=EM;
    else
        em=EM;
    Feature1[0]=ed;
    Feature1[1]=me;
    Feature1[2]=em;

    ME = (float)(m_RightEye.x -m_MidMouth.x)*(float)(m_RightEye.x -m_MidMouth.x);
    ME+=(float)

```

```

(m_LeftEye.y-m_MidMouth.y)*(float)(m_LeftEye.y-m_MidMouth.y);
    ME= sqrt((float)ME);
    if(me)
        mel=ME;
    else
        me=ME;
    Feature2[0]=ed1;
    Feature2[1]= mel;
    Feature2[2]= em1;

    m_bMidNoseOK = TRUE;
    m_bLeftNostrilOK = TRUE;
    m_bRightNostrilOK = TRUE;
    m_bMatchOK=TRUE;
}
//////////////////////////////////////
//人脸匹配
//////////////////////////////////////
void CFaceDetectDlg::OnBtnMatch()
{
    // TODO: Add your control notification handler code here

    if(!m_bMatchOK)
    {
        AfxMessageBox("请预先先后打开两幅图像并进行操作!");
    }
    else
    {
        float score1,score2,score3;
        score1=fabs(Feature1[0]-Feature2[0]);
        score2=fabs(Feature1[1]-Feature2[1]);
        score3=fabs(Feature1[2]-Feature2[2]);

        if ((score1<0.1)&&(score2<0.1)&&(score3<0.1))
        {
            AfxMessageBox("先后两幅图片中为同一个人!");
        }
        else
        {
            AfxMessageBox("先后两幅图片中为不同的人!");
        }
    }
    MakeBitMap();
}

```

## 11.5 经验分享

本章的人脸识别和第 10 章的指纹识别都属于生物特征识别技术范畴。生物特征识别技术所研究的生物特征包括人脸、指纹、手掌纹、掌型、虹膜、视网膜、静脉、声音（语音）、体形、红外温谱、耳型、气味、个人习惯（例如敲击键盘的力度和频率、签字、步态）等，相应的识别技术包括：人脸识别、指纹识别、掌纹识别、虹膜识别、视网膜识别、静脉识别、语音识别（用语音识别可以进行身份识别，也可以进行语音内容的识别，只有前者属于生物特征识别技术）、体形识别、键盘敲击识别、签字识别等，其中除语音识别和键盘敲击识别外，均涉及数字图像处理与分析技术，这类识别方法的基本步骤都是先检测和定位识别目标，然后做特征提取，最后进行相似性判断。

特征是图像分析中的一个核心要素。目标检测时，依赖的是某一类概念中所有个体共有的特征（共同点）；而个体识别时，依赖的是各个个体取值不同的特征（差异点）。选择的特征不同，得到的识别结果可能会截然不同。

本章通过一个简单的演示系统介绍了人脸检测和识别的基本原理及编程实现方法，实际工程应用中，除了要在界面设计上下工夫以求更方便用户使用外，还要根据任务背景和具体需求仔细研究并选择鲁棒性更强的人脸特征，以抵抗光照、姿态、遮挡、表情和年龄等多种复杂因素的干扰，从而达到更好的检测与识别结果。

## 第 12 章 运动车辆检测跟踪系统

“千车万马九衢上”、“车如流水马如龙”，无论是描写长安街头的忙碌，还是金陵上苑的盛大，车水马龙已成为诗人描写繁华景象的经典桥段。时至今日，“车如流水”应犹在，“马如游龙”却难寻了。自从 1886 年德国工程师 Karl Benz 为其制造的三轮汽车申请专利至今，一百多年过去了，汽车逐渐成为现代生活中不可或缺的交通运输工具，但交通拥堵、交通安全等一系列问题也随着车辆的增多接踵而来，对交通运输管理提出了挑战。为了大范围、全方位、实时、准确、高效地进行交通运输管理，人们开发了智能交通系统（Intelligent Transportation Systems, ITS）。ITS 中首要的问题是交通信息的采集和处理，包括车辆的检测与跟踪等。本章围绕一个实例介绍基于视频的运动车辆检测跟踪的技术原理及编程实现方法。

**本章要点：**

- 运动目标检测技术
- 运动目标跟踪技术
- 运动车辆检测跟踪系统功能描述
- 运动车辆检测跟踪系统的总体结构和主要流程
- 运动车辆检测跟踪系统的编程实现

### 12.1 核心技术原理

运动车辆检测跟踪系统的核心是车辆的快速检测和定位，只有准确地检测和定位出车辆才能进行实时的跟踪。系统涉及的核心技术主要有运动目标检测技术和运动目标跟踪技术。

#### 12.1.1 运动目标检测技术

日常生活中，如活动的人、动物和行驶的汽车等都是常见的运动目标，利用图像捕获并跟踪感兴趣的运动目标在许多领域有着广泛的应用。在现实生活中，大量有意义的视觉信息包含在运动之中，运动目标检测与跟踪技术的应用场合十分广泛，比如交通流量的监测、重要场所的安保、航空和军用飞行器的制导、汽车的自动驾驶或辅助驾驶等。目前，视频信号的智能化处理需求日

益增加, 正确地从视频流中提取运动目标是许多智能视频监视系统, 如: 视频监视、交通自动监控、人脸检测与跟踪等的基础部分。运动目标检测和识别是计算机视觉中的一个基础的、关键的任务, 它被看做是视觉系统的一个重要功能。

运动目标检测的目的是将序列图像中的运动目标从背景图像中检测并分割出来。关于运动目标检测的研究大致可分为以下两类: 一是摄像头随着运动目标移动, 始终保持目标在图像的中心附近, 如在导航系统和辅助驾驶系统中, 摄像头被安装在它的主体(如车、飞机等)上面, 摄像头随着运动目标移动; 二是摄像头相对处于静止状态, 只对视场内的目标进行检测、定位, 如监视某一路口车流量等的固定摄像头。目前大多数目标跟踪算法研究的都是第二种情况。

光照的变化、背景混乱运动的干扰、运动目标的影子、摄像机的抖动, 以及运动目标的自遮挡和互遮挡现象都给运动目标的正确检测带来了极大的挑战。同时由于运动目标的正确检测与分割效果极大地影响运动目标的跟踪和分类, 因此运动目标检测成为计算机视觉研究中一项重要课题。

目前常见的运动目标的检测方法主要有帧差分法、背景差分法和运动场估计法。

### 1. 帧差方法

帧差方法的基本思想是: 将相邻两帧(或多帧)连续图像逐像素相减, 以去除静止或移动缓慢的物体及背景, 获取差值图像, 并对该差值图像进行二值化, 若像素值大于某一阈值, 则判定此像素出现在运动目标上, 见公式(12-1)。且经过阈值操作后得到的图像直接显示出了目标的位置、大小和形状等信息, 从而达到检测出运动目标的目的, 它是消除两帧连续图像中的静止物体以及提供运动物体(车辆)运动轨迹最直接的方法。除了最简单的逐像素相减, 帧差图像还可以由两组属于相邻图像帧的像素(如相邻的四个元素)的均值相减得到。

$$d(x, y) = \begin{cases} 0 & \text{如果 } |f_1(x, y) - f_2(x, y)| < Th \\ 1 & \text{其他} \end{cases} \quad (12-1)$$

式中  $f_1(x, y)$  为输入图像帧,  $f_2(x, y)$  为背景图像帧。如果输入图像帧不含目标, 则和背景图像帧几乎相同, 此时  $d(x, y)$  为 0, 相反, 如果输入图像帧中包含目标, 则和背景图像帧有很大不同, 此时根据阈值  $Th$  的大小判定  $d(x, y)$  的值。

帧差方法的优点是计算简单且不易受环境光线变化的影响, 但它不能检测静止车辆, 且处理效果与图像采样频率以及被检测车辆的车速有关。如果视频检测器采样频率过小, 而车速较快, 可能会造成误分割, 这种情况如图 12-1 所示。

反之, 如果采样频率过大且车速较慢, 又会造成过度覆盖, 极端情况下运动物体可能完全重叠, 类似于静止车辆, 从而导致无法分割出运动物体, 这种情况如图 12-2 所示。



图 12-1 车速过快造成误分割

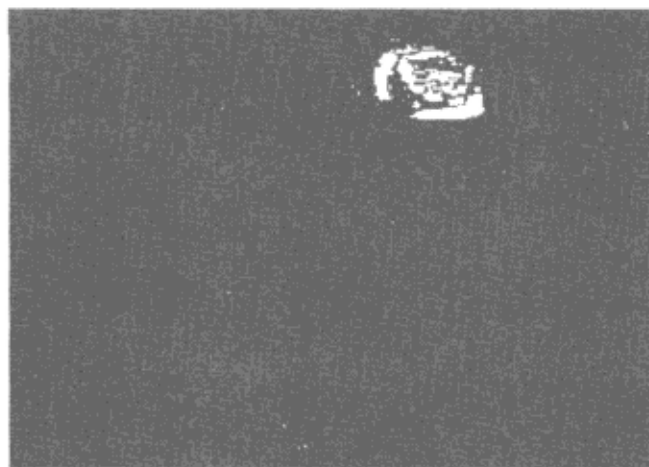


图 12-2 车速过慢造成误分割

## 2. 背景差分法

背景差分法是运动目标检测最常用的方法之一，它的基本思想是：通过背景建模，利用相邻序列图像估计视频中不变的或有规律变化的背景，将输入图像与背景图像进行比较，从中分割出前景运动目标。背景图像通常是整个场景中变化相对缓慢的部分，如交通视频图像中的道路（与移动的车辆相比它的变化相对缓慢）。

背景差分法的另一个比较重要的内容就是背景图像的获得。由于图像很容易受到外界因素的干扰，如光照、阴雨、摄像机的抖动等，都会对差分的效果产生很大影响，因此如何得到一个好的背景至关重要。背景图像的获得主要有以下三种方法：

- 通过求多幅图像的像素点均值得到。
- 通过求多幅图像的像素点中值得到。
- 通过建立自适应模型得到。

现在普遍采用的背景模型是基于时间的多幅图像平均法，通过求一段连续时间内多幅图像累加和的平均值而得到背景图像。如公式（12-2）所示。

$$B_t = \frac{1}{n} \sum_{i=1}^n I_i \quad (12-2)$$

其中， $B_t$  为  $t$  时刻的背景图像帧， $I_i$  为  $t$  时刻的输入图像帧， $n$  为帧的总数。

与帧差方法相比，背景差分法可以检测出短时间静止的车辆（长时间静止的车辆可以归为背景），也不受车速快慢的限制，而且，背景差分法可以通过简化算法降低计算量，满足视频检测的实时性要求。虽然背景消减法可以提取出完整的目标图像，但在实际应用中仍有许多问题需要解决，实际应用中一幅标准的背景图像总是不容易得到的，一种简单的获取背景图像的方法是当场景中无任何目标出现时就捕获背景图像，很显然随着时间的推移，外界的光线会变化，这会引起背景图像的变化，因而这种人工的非自适应的方法获得的背景图像，会随着时间的推移，对场景中光照条件、大面积运动和噪声比较敏感，出现许多伪运动目标点，不利于目标的准确检测，



如图 12-3 所示,有许多干扰噪声出现。

本章介绍的运动车辆检测跟踪系统采用的是背景差分法,背景图像采用基于时间的帧平均法得到,具体过程如下:

- 1) 从视频中获取一帧。
- 2) 将帧进行高斯平滑。
- 3) 对帧进行累加。
- 4) 将最后的累加帧总和进行平均。

具体累加的帧数可由读者自己设定,累加前将帧进行高斯平滑(模糊处理),这样做的目的是为了克服由于摄像机的抖动而造成的背景轻微摇摆,同时也克服了单个像素间无联系的毛病,小片像素相联系增加了背景的健壮性。

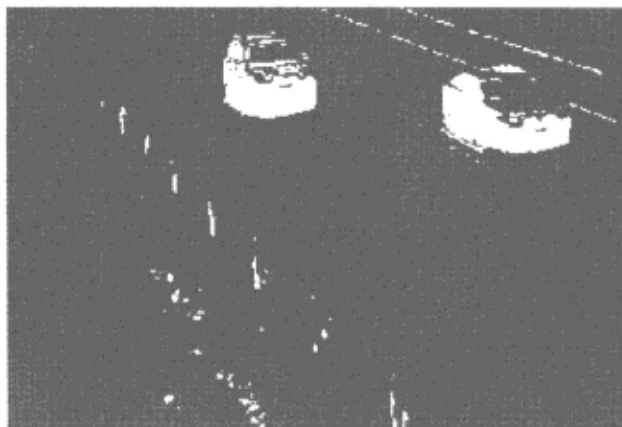


图 12-3 噪声干扰

### 3. 运动场估计法

运动场估计法是指通过视频序列的时空相关性来分析估计运动场,建立相邻帧间的对应关系,进而利用目标与背景表现运动模式的不同进行运动目标的检测与分割。主要是光流法,光流法检测采用了目标随时间变化的光流特性,当物体运动时,在图像上对应物体的亮度模式也在运动,从而称光流是图像亮度模式的表现运动。在光流场中,不同的物体会有不同的速度,大面积背景的运动会在图像上产生较为均匀的速度矢量区域,这为具有不同速度的其他运动物体的检测提供了方便。

值得注意的是,有些光流计算结果只在物体边缘与背景有较明显的速度差,而在比较均匀的物体内部则差别不明显,因而,简单的速度分割之后,还需要一定的后处理工作才能完整地检测出运动物体。光流法的优点是能够检测独立运动的对象,不需要预先知道场景的任何信息,并且可用于摄像机运动的情况,对目标在帧间的运动的限制较少,可以处理大的帧间位移。但多数光流法很耗时,计算也相当复杂,并且抗噪性能差,除非有特殊的硬件支持,否则很难实现实时检测。

## 12.1.2 运动目标跟踪技术

运动目标跟踪就是在视频图像序列中实时地发现并标记运动目标,在帧与帧之间建立车辆运动的某些特征,如位置、速度、形状和方向等之间的联系,不断跟踪目标,并计算出运动目标的轨迹。

目前运动目标跟踪的方法有很多,针对视频图像中运动目标的跟踪方法主要有以下几种:

### 1. 基于模型的跟踪

在基于模型的目标跟踪算法中,可以表达运动目标的方法有以下三种:

- 线图法 (Stick Figure): 车辆的运动是刚体运动, 该表达方法将车辆以直线来近似。
- 二维轮廓 (2-D Contour): 该表达方法使用车辆的轮廓进行运动的分析。
- 立体模型 (Volumetric Model): 利用三维模型来描述车辆的结构细节, 因此需要用到较多的计算参数, 匹配过程中计算量较大。

基于立体模型的跟踪方法被应用于估计运动目标的运动, 从二维图像中推断出运动目标的立体形状。基于立体模型的跟踪方法的显著优点是即使在复杂驾驶操作、明显交通阻塞的情况下利用模型知识的结果会鲁棒地得到跟踪结果; 最大的缺点是由于计算的工作量大, 实时性差。虽然模型法能够获得对图像内容的理解, 但建立摄像机模型时需要测量详细的摄像机与交通场景之间的空间几何特征, 这使得这种方法在实际应用中存在着很大的局限性。同时模型法的稳健性也不够高, 当摄像机由于外力原因产生微小角度变化时就可能造成检测失败, 且对遮挡情况下的车辆也会发生误检。

## 2. 基于区域的跟踪

区域跟踪的思想是把每个运动物体与某个运动区域相联系, 然后对该区域进行跟踪。区域跟踪实现较为简单, 在许多系统中得到了广泛的应用。用基于区域跟踪的方法来跟踪车辆, 对于城市交通中存在的遮挡问题不太敏感, 而且用这种方法跟踪可以改善图像的分割。基于区域的车辆跟踪方法由于在跟踪目标之前能完成最后的图像分割, 因此在它们的形状突然改变或消失时都能正确分割图像。但是某些体积较大的车辆会产生较大的影子, 遮挡一些体积较小的车辆, 这对车辆跟踪是很不利的。

## 3. 基于动态轮廓的跟踪

基于动态轮廓的跟踪思想是利用封闭的曲线轮廓来表达运动目标, 并且该轮廓能够自动连续地更新。相对于基于区域的跟踪方法, 轮廓表达有计算复杂度小的优点, 如果开始阶段能够合理地分开每个运动目标并实现轮廓初始化, 即使在有部分遮挡存在的情况下也能连续地进行跟踪, 然而轮廓初始化通常是很困难的。

## 4. 基于特征的跟踪

由于运动目标的个体特征有许多, 不管是刚体运动目标, 还是非刚体运动目标, 在序列图像中相邻的两帧图像, 由于图像序列间的采样时间间隔很小, 可以认为这些个体特征在运动形式上具有平滑性, 因此可以用直线、曲线、参照点等个体特征来跟踪运动目标。但是提取运动目标的主要特征有时是很困难的。

本章介绍的运动车辆检测跟踪系统采用的是基于动态轮廓的跟踪方法, 由于车辆在运动中外界环境会不断变化, 车辆的轮廓也会出现不规则变形, 有时还会出现各种断裂不连续的现象, 甚至内部有空洞, 如图 12-4 中最上方的那个不规则轮廓。考虑到车辆是长方形, 为了更好地跟踪车

辆运动轨迹，采用了中心点的思想，对车辆边缘轮廓上的所有点进行统计，求出这些点的中心点（坐标中点），用中心点及其周围（上、下、左、右）四个点来表示运动物体。运动车辆轨迹图便是根据中心点位置的移动绘制的。

## 12.2 系统功能

运动车辆检测跟踪系统主要包括两项基本功能，即运动车辆检测和运动车辆跟踪。

### 12.2.1 功能描述

运动车辆检测跟踪系统主要处理的是 AVI 格式的视频文件，若对视频部分稍加修改也可直接处理由数字摄像机拍摄的视频，进行实时场景的运动跟踪。该程序主要有以下几个功能：

- 1) 读入 AVI 格式的视频文件，并得到每帧图像。
- 2) 对视频前几帧进行处理，获得背景图像。
- 3) 通过背景做差法获得目标前景图像。
- 4) 对前景图像进行处理，以达到目标检测和分割的目的。
- 5) 对分割出来的目标进行实时跟踪，并绘制出运动目标的轨迹。

### 12.2.2 界面效果

运动车辆检测跟踪系统启动的主界面如图 12-5 所示。显示原始视频和提取背景图像的窗口分别如图 12-6 和图 12-7 所示。



图 12-4 提取的运动目标不规则轮廓



图 12-5 运动车辆检测跟踪系统主界面



图 12-6 原始视频

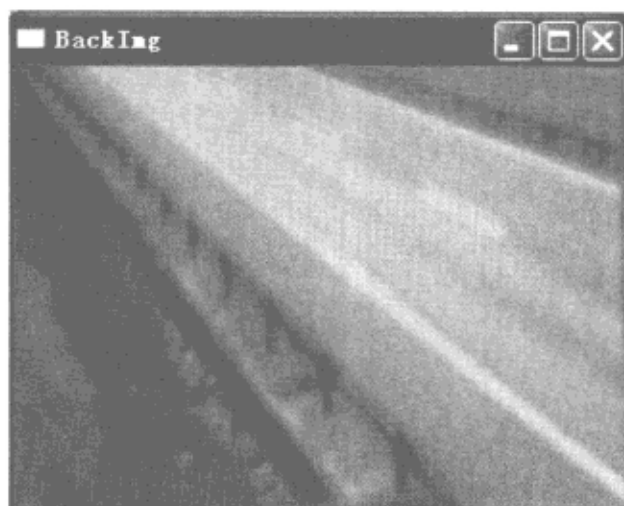


图 12-7 提取的背景图像

车辆跟踪的界面如图 12-8 所示。左侧显示的是视频图像，右侧显示的是对车辆目标的跟踪情况，其中白色点代表车辆中心点。绘制的车辆运动轨迹如图 12-9 所示。



图 12-8 车辆及对应的目标图像



图 12-9 车辆运动轨迹图

## 12.3 系统结构与流程

运动车辆检测跟踪系统架构如图 12-10 所示。首先由图像采集设备得到实时交通场景的视频序列图像，然后将序列图像传送到车辆检测模块进行车辆检测和分割提取，而后对识别出的车辆进行跟踪，最后根据跟踪的结果绘制出车辆的运动轨迹。



图 12-10 运动车辆检测跟踪系统架构图

### 12.3.1 总体结构

运动车辆检测跟踪系统共包括 5 个模块，其总体结构如图 12-11 所示。

### 12.3.2 主要流程

运动车辆检测跟踪系统虽然载入的是视频文件，但实质处理的是视频中的帧图像，首先从视频中读取一帧，然后判断帧是否有效，若无效则程序结束，若有效则对帧图像进行处理，直至视

频播放结束，系统的主要流程如图 12-12 所示。

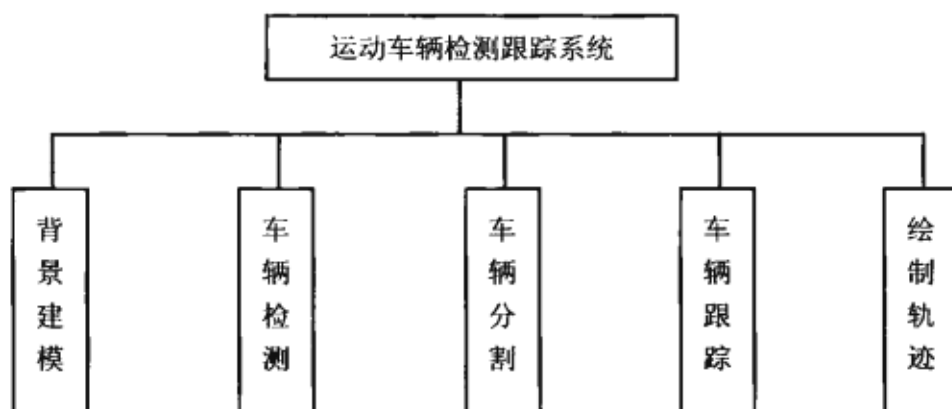


图 12-11 运动车辆检测跟踪系统总体结构

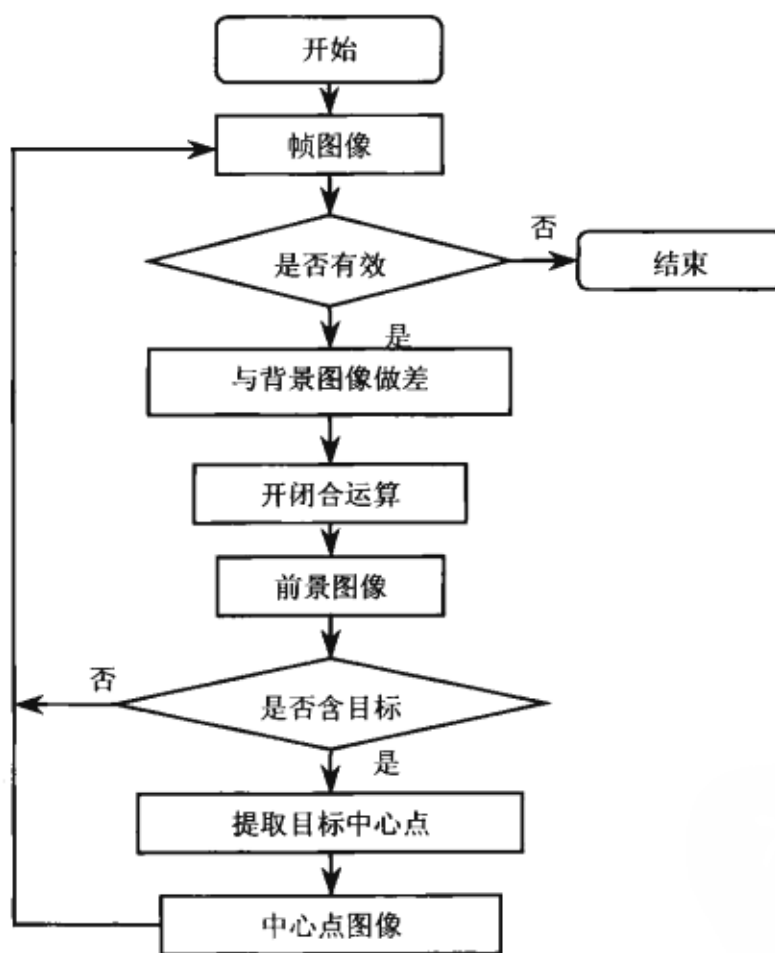


图 12-12 运动车辆检测跟踪系统流程图

## 12.4 编程实现

运动车辆检测跟踪系统采用 Visual Studio 2008 开发平台结合 OpenCV 函数库编程实现。



### 12.4.1 变量定义模块

运动车辆检测跟踪系统用到的主要变量定义如下:

```
//用于 OpenCV 的变量的定义
CvCapture * pCapture;           //定义获取视频播放结构
CvCapture * pCapture2;
IplImage * pFrame;             //定义视频帧
IplImage * pFrImg;             //前景图像
IplImage * pBkImg;             //背景图像
IplImage * pCenterImg;         //中心点图像
IplImage * TrackImg;           //轨迹图像
int count_center_point;         //统计中心点的个数
CvPoint cen_point[1000000];     //定义轨迹点数组, 用来存放中心点

//变量的初始化
CvCapture * pCapture=NULL;
CvCapture * pCapture2=NULL;
IplImage * pFrame=NULL;
IplImage * pFrImg=NULL;
IplImage * pBkImg=NULL;
IplImage * pCenterImg=NULL;
IplImage * TrackImg=NULL;
count_center_point=0;
memset(cen_point,0,sizeof(cen_point)); //将数组的初始值都赋值为 0
```

### 12.4.2 文件打开模块

由于系统是基于 MFC 的单文档应用程序, 故文件的打开模块采用的是基于 Windows 的对话框模式。

文件打开模块的代码如下所示:

```
void CMoveTrackingDoc::OnFileOpen()
{
    // TODO: 在此添加命令处理程序代码
    //文件类型过滤, 只打开 AVI 格式的文件
    LPCTSTR lpszFilter="AVI Files(*.avi)|*.avi|任何文件|*.*||";
    //打开文件对话框
    CFileDialog dlg1(TRUE,lpszFilter,NULL,
        OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,lpszFilter,NULL);
    if(dlg1.DoModal()==IDOK)
    {
        //初始化 Capture 结构
        if(!(pCapture=cvCaptureFromFile(dlg1.GetPathName()))))
        {
            AfxMessageBox("无法打开文件!");
        }
    }
}
```

```

    }
    //保存一份副本,方便以后使用
    pCapture2=cvCaptureFromFile(dlg1.GetPathName());
}
else
    AfxMessageBox("打开文件失败!");
}

//*****显示视频播放窗口和视频的第一帧*****
cvNamedWindow("video", 1);           //视频播放窗口
cvMoveWindow("video", 120, 150);      //移动窗口,使窗口排列有序
//显示第一帧图像
CvCapture * pCapture3;
IplImage *pFirst_Frame;
pCapture3=cvCaptureFromFile(dlg1.GetPathName());
pFirst_Frame=cvQueryFrame(pCapture3);
cvShowImage("video",pFirst_Frame);
cvWaitKey(0);

```

### 12.4.3 背景提取模块

背景图像是采用基于时间的连续多帧求平均法提取的。背景提取程序的代码如下:

```

void CMoveTrackingDoc::OnBackobtain()
{
    // TODO: 在此添加命令处理程序代码
    //定义程序用到的数据
    IplImage *pAvgImg=NULL;           //均值累加图像
    IplImage *pBkGrayImg=NULL;        //背景灰度图像
    IplImage *pScratchImg;             //临时转换中间图像
    Int nFrmNum=0;                     //统计视频帧数

    //定义视频播放窗口
    cvNamedWindow("video", 1);
    //移动窗口,使窗口排列有序
    cvMoveWindow("video", 50, 20);

    //显示背景图像窗口
    cvNamedWindow("BackImg",1);
    cvMoveWindow("BackImg", 450,20);

    //播放视频,统计背景帧数,求出背景
    while(nFrmNum<=140)               //统计视频的前140帧
    {
        //从视频结构中按顺序获得一帧图像
        pFrame=cvQueryFrame(pCapture);
        if(!pFrame)                    //判断帧是否有效
            break;
    }
}

```

```

        nFrmNum++; //统计帧数
        //如果是第一帧, 需要申请内存, 并初始化
        if (l==nFrmNum)
        {
            pBkImg=cvCreateImage(cvSize(pFrame->width, pFrame->height),8,1);
            pBkGrayImg=cvCreateImage(cvSize(pFrame->width, pFrame->height),8,1);
            pAvgImg=cvCreateImage(cvSize(pFrame->width,pFrame->height),IPL_DEPTH_32F,1);
            pScratchImg=cvCreateImage(cvSize(pFrame->width,pFrame->height),IPL_DEPTH_32F,1);
        }
        else
        {
            //将帧图像转化为灰度图像
            cvCvtColor(pFrame,pBkGrayImg, CV_BGR2GRAY);
            //将灰度图像进行高斯平滑
            cvSmooth(pBkGrayImg,pBkGrayImg,CV_GAUSSIAN,5,0,0);
            //进行累加前, 对图像进行格式转换
            cvCvtScale(pBkGrayImg,pScratchImg,1,0);
            //进行图像的累加
            cvAcc(pScratchImg,pAvgImg);
            //在视频窗口中显示帧图像, 播放视频
            cvShowImage("video",pFrame);
            //为了获得更好的观察效果, 人为减慢视频播放速度
            char c=cvWaitKey(150);
            if(c==27) //按下 Esc 键退出视频播放
                break;
        }
    }
    //用得到的图像总和除以图像个数, 注意第一帧不包含在内
    cvConvertScale(pAvgImg,pAvgImg,(double)(1.0/(nFrmNum-1)));
    //得到视频图像背景
    cvCvtScale(pAvgImg,pBkImg,1,0);
    //在固定窗口内显示背景图像
    cvShowImage("BackImg",pBkImg);

    cvWaitKey(0); //等待按键结束
    cvDestroyWindow("video"); //销毁窗口
    cvDestroyWindow("BackImg");
}

```

#### 12.4.4 车辆跟踪与检测模块

车辆的检测和跟踪是合并在一个程序模块中编写的, 因为这样可以加快系统的运算速度, 节省内存开销, 其源代码如下:

```

void CMoveTrackingDoc::OnCartrack()
{
    // TODO: 在此添加命令处理程序代码
}

```

```

IplImage *pCannyImg=NULL;    //定义边缘检测图像
IplImage *pFrame2=NULL;      //定义视频播放帧
IplImage *pCenterImg=NULL;
float count_X=0.0;           //统计 X 坐标的总和
float count_Y=0.0;           //统计 Y 坐标的总和
int count_point=0;
cvNamedWindow("video",1);
cvMoveWindow("video",50,20);
cvNamedWindow("Track",1);
cvMoveWindow("Track",450,20);

pFrImg=cvCreateImage(cvSize(pBkImg->width, pBkImg->height),8,1);
pCannyImg=cvCreateImage(cvSize(pBkImg->width, pBkImg->height),8,1);
pCenterImg=cvCreateImage(cvSize(pBkImg->width, pBkImg->height),8,1);
while(1)
{
    pFrame2=cvQueryFrame(pCapture2); //从视频结构中按顺序获得一帧图像
    if(!pFrame2) //判断帧是否有效
        break;
    cvCvtColor(pFrame2, pFrImg, CV_BGR2GRAY); //将帧图像转化为灰度图像
    cvSmooth(pFrImg, pFrImg, CV_GAUSSIAN, 5, 0, 0); //将灰度图像进行高斯平滑
    //将当前帧和背景图像做绝对值差, 求得前景图像
    cvAbsDiff(pFrImg, pBkImg, pFrImg);
    //将前景图像二值化
    cvThreshold(pFrImg, pFrImg, 100, 255, CV_THRESH_BINARY);

    //定义腐蚀和膨胀用的核
    IplConvKernel * kernel_5_5;
    kernel_5_5=cvCreateStructuringElementEx(5, 5, 2, 2, CV_SHAPE_ELLIPSE, 0);
    //对前景图像进行开闭运算以除去杂点, 分割出运动物体
    //对图像进行腐蚀运算
    cvErode(pFrImg, pFrImg, kernel_5_5, 1);
    //对图像进行膨胀运算
    cvDilate(pFrImg, pFrImg, kernel_5_5, 1);

    //对目标图像进行 Canny 边缘检测
    cvCanny(pFrImg, pCannyImg, 80, 2, 3);
    //计算目标图像的中心点
    for(int j=0; j<pCannyImg->height; j++)
    {
        uchar * ptr=(uchar *) (pCannyImg->imageData+j*pCannyImg->widthStep);
        for(int i=0; i<pCannyImg->width; i++)
        {
            if(ptr[i]>0)
            {
                count_point++;
                count_X+=i;
            }
        }
    }
}

```

```

        count_Y+=j;
    }
}
cvZero(pCenterImg); //将中心点图像初始化为黑色
if(count_point>=5) //说明有运动车辆, 进行跟踪
{
    //计算图像的中心点, 为后面的轨迹绘制做准备
    cen_point[count_center_point].x=int(count_X/count_point);
    cen_point[count_center_point].y=int(count_Y/count_point);
    count_center_point++;

    //将中心点及中心点上下左右的五个点设置为白色, 来标记运动物体
    CV_IMAGE_ELEM(pCenterImg,uchar,int(count_Y/count_point),
        int(count_X/count_point))=255;
    CV_IMAGE_ELEM(pCenterImg,uchar,int(count_Y/count_point)-1,
        int(count_X/count_point))=255;
    CV_IMAGE_ELEM(pCenterImg,uchar,int(count_Y/count_point)+1,
        int(count_X/count_point))=255;
    CV_IMAGE_ELEM(pCenterImg,uchar,int(count_Y/count_point),
        int(count_X/count_point)+1)=255;
    CV_IMAGE_ELEM(pCenterImg,uchar,int(count_Y/count_point),
        int(count_X/count_point)-1)=255;
}
cvShowImage("video",pFrame2);
cvShowImage("Track",pCenterImg);
//将变量重新设置为初始值
count_point=0;
count_X=0;
count_Y=0;
char c=cvWaitKey(150); //控制播放速度
if(c==27) //按下 Esc 键退出视频播放
    break;
}
}

```

### 12.4.5 轨迹绘制模块

绘制车辆的运动轨迹是系统的最终目标。在取得物体中心点的过程中, 已经将一系列的中心点都保存到了一个专门记录车辆运动轨迹的数组中, 当处理完视频后, 数组里就记录了完整的车辆运动轨迹。只要扫描整个数组, 按照里面的数值把曲线绘制出来即可, 为了保证轨迹和物体实际运动路线的对比效果, 运动轨迹的背景图像和视频图像的大小一致。轨迹绘制代码如下:

```

void CMoveTrackingDoc::OnTrackpaint()
{
    // TODO: 在此添加命令处理程序代码
}

```



```

TrackImg=cvCreateImage(cvGetSize(pBkImg),8,3);
cvZero(TrackImg);
CvScalar color={0,255,0}; //设置轨迹颜色
//绘制轨迹曲线
for(int i=0;i<count_center_point-1;i++)
{
    cvLine(TrackImg,cen_point[i],cen_point[i+1],color);
}
cvNamedWindow("Track",1);
cvShowImage("Track",TrackImg);
cvWaitKey(0);
cvDestroyWindow("Track");
}

```

## 12.5 经验分享

运动车辆检测跟踪系统的开发环境是 Visual Studio 2008, 因为它能和 OpenCV 很好地结合, 选择 MFC 框架是因为其独一无二的良好窗口界面, 开发语言选用 C++ 是考虑到运动跟踪系统要求有很高的实时性, 而选用 OpenCV 函数库是因为其简洁、高效的特点。它把图像处理中常用的一些基本运算进行了高效的封装, 提供了很好的函数接口, 当要对图像进行某项处理时只需调用该函数即可, 而不用具体地去考虑内部的运算细节, 这极大地方便了初学者, 也提高了编程效率, 同时也便于开发人员对程序进行调试。但要注意 OpenCV 中一些数据定义及使用的一些细节, 否则会出现一些莫名其妙的错误, 比如在开发过程中经常用到图像数据结构 IplImage, 当定义完一个该类型的变量 (如 SrcImage) 后, 一定要用 cvCreateImage() 函数在内存中创建出这个对象, 否则当后面用到它的时候就会出错。

本章给出的运动车辆检测跟踪系统主要是演示车辆检测和跟踪的基本原理, 故系统在设计时省略了一些基本的图像操作的效果演示, 如图像灰度化、背景做差、二值化、膨胀、腐蚀等。若想直观地看到这些中间处理结果, 可以在源代码的相应位置自行插入 cvShowImage() 函数显示中间结果。

车辆的检测和跟踪功能也经常分开使用。比如, 车辆检测可用于交通流量信息统计, 车辆跟踪可用于自动识别违章行驶行为, 等等。应用是五花八门的, 但核心技术的原理都大同小异。在做实际工程项目时, 主要是利用这些基本原理结合具体任务要求重新设计程序结构、流程和界面。此外, 在一些实际系统中, 往往都是直接连接视频采集设备, 自动读取视频数据, 自动显示监控画面, 这些都要从方便用户使用的角度去设计。



## 第 13 章 车型识别系统

元代医官许国桢之母韩氏，因烧得一手好菜毛遂自荐到后宫当了厨师，她根据忽必烈和东西宫娘娘各自的喜好，为他们烹制不同的菜肴，并给每道菜取一个投皇上和娘娘所好的名字，人们把韩氏这种做法叫“看人下菜碟儿”，表示对不同的人给予不同的待遇。在收费站、停车场、洗车行等地对车辆收取费用时，也是看“车”下菜碟的，对不同的车型收取不同的费用。随着智能交通系统的推广，这些场合陆续开始实行自动收费，于是车型的自动识别技术应运而生。本章介绍车型识别的核心技术原理及编程实现方法。

**本章要点：**

- 基于背景去除的目标分割技术
- 车型特征提取技术
- 车型分类识别技术
- 车型识别系统功能描述
- 车型识别系统的基本结构和主要流程
- 车型识别系统的编程实现

### 13.1 核心技术原理

车型识别系统涉及的核心技术包括基于背景去除的目标分割技术、车型特征提取技术和车型分类识别技术。

#### 13.1.1 基于背景去除的目标分割技术

车型识别的第一步就是从图像中分割出目标车辆，然而摄像机拍摄到的图像往往都有复杂的背景，如何从复杂的背景下分割出完整的目标车辆区域是整个系统的前提和基础，其分割质量的好坏直接影响到最后车型分类的准确性。

由于车辆的类型繁多，没有一种固定的颜色或纹理模式，使得直接采用区域分割方法较难实现；又由于摄像头和运动车辆之间的相对位置会发生变化，因此目标车辆在图像中出现的位置、

方向角度及其大小都是无法预知的，这也增加了目标车辆区域提取的难度。此外，光照、阴影干扰及背景变化等也是必须要注意的因素。

考虑到上述问题，为了增强系统的实用性，本章介绍的车型识别系统采用了背景消减法来去除背景，下面以图 13-1 的背景图像和图 13-2 的带前景图像为例介绍分割目标车辆的整个过程。

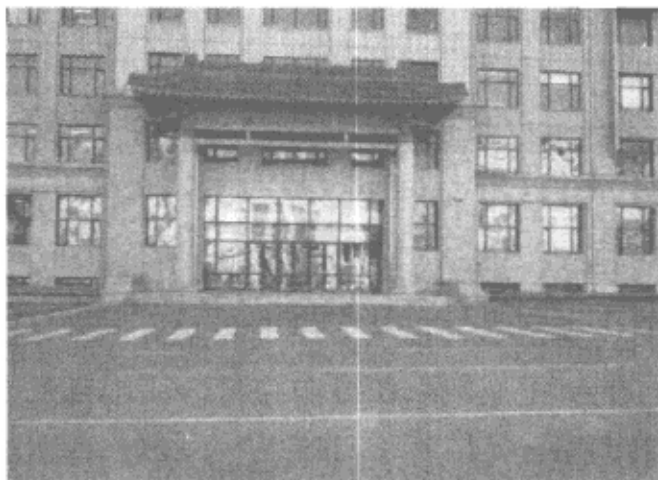


图 13-1 背景图像



图 13-2 带前景图像

### 1. 背景消减法

背景消减法的思想是将含有运动物体的图像与事先已经获得的背景图像相减得到运动部分。只用背景消减法并不能完全分割出目标车辆区域，但是仍然可以得到有用的分割依据。背景部分的灰度变化是缓慢的，而目标车辆区域所对应的位置在两幅图像中的变化是剧烈的，这种变化反映在差分图像中就是：背景部分虽然不能完全消除，但灰度值较小，而目标车辆区域的灰度值则较大，这就为目标车辆区域的分割提供了依据。

拍摄图像时由于光照变化和相机抖动等因素的干扰，两幅图像的背景有轻微的位置偏移，直接进行做差运算并不能有效地去除背景。为此，在图像做差前，要先对两幅图像进行“平滑处理”（也称“模糊处理”）。本章采用高斯平滑处理，这样做的目的是为了消除单个像素间无联系的问题，尽量削弱背景位置偏移造成的影响。做差运算的数学公式见公式（13-1），图 13-3 和图 13-4 分别是直接做差运算和先平滑后做差运算的图像。

$$d(x, y) = |f_1(x, y) - f_2(x, y)| \quad (13-1)$$

式中  $f_1(x, y)$  为前景图像， $f_2(x, y)$  为背景图像，目标图像的像素值  $d(x, y)$  为背景和前景图像对应坐标点的像素值相减后取绝对值得到的。

将图 13-3 和图 13-4 两幅图进行比较可以看出后者的背景像素变淡，而车身比较亮，汽车轮子的轮廓也清晰可见。

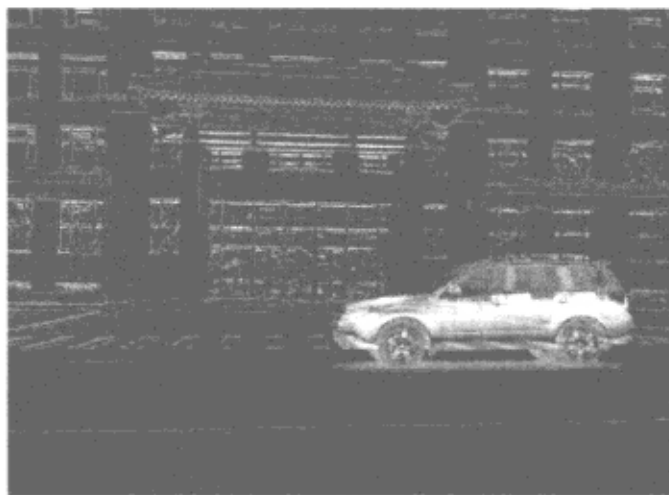


图 13-3 直接做差得到的图像

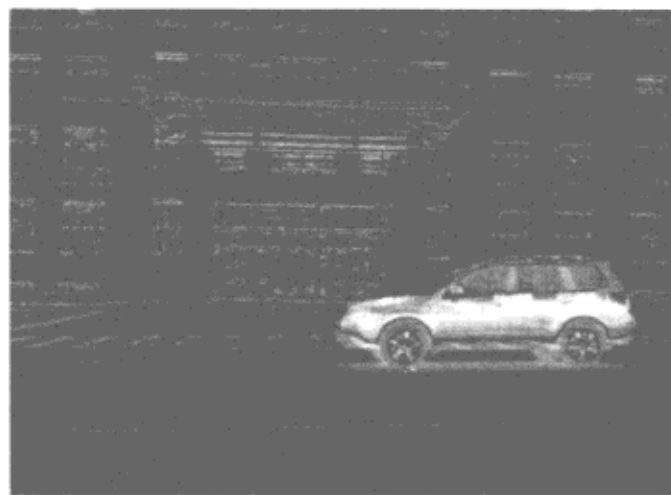


图 13-4 先平滑后做差得到的图像

## 2. 图像二值化

在进行了背景做差运算后，图像中的背景并没有完全消除，但从图像中我们看出背景区域已经变得比较暗淡，而目标车辆区域则比较明亮。为了能够把车辆从图像中完全分割出来还要对图像进行二值化处理。图像二值化就是把图像中的像素根据一定的标准分成两种颜色。在车型识别系统中是根据像素的灰度值处理成黑（像素值为 0）和白（像素值为 255）两种颜色，如公式 (13-2) 所描述的，这样就基本上能把目标和背景分割开来，本章采用了阈值分割法来实现图像的二值化。

$$g(x, y) = \begin{cases} 0 & f(x, y) < T \\ 255 & f(x, y) \geq T \end{cases} \quad (13-2)$$

其中， $f(x, y)$  为待分割的图像， $g(x, y)$  为分割后的图像， $T$  为阈值。

阈值分割法计算简单，而且总能用封闭且连通的边界定义不交叠的区域，对目标与背景有较强对比度的图像显示出较好的分割效果。这种方法的一个关键技术是最优阈值的确定，如果阈值选取过高，则过多的目标区域将被划分为背景区；反之，如果阈值选取过低，则过多的背景区域将被划分为目标区。

本章的车型识别系统采用的是最大方差阈值分割法，也叫 Otsu 法，是一种使用类间方差最大值的自动确定阈值方法，该方法是在判别分析技术的基础上推导出来的，效果较好，其基本原理如下。

设图像的像素总数为  $N$ ，灰度范围为  $[0, L-1]$ ，灰度值  $i$  的像素数为  $n_i$ ，则  $i$  的概率为

$$p_i = \frac{n_i}{N} \quad (13-3)$$

把图像中的像素按灰度值用阈值  $T$  分成两类  $C_0$  和  $C_1$ ， $C_0$  对应于灰度值在  $[0, T-1]$  之间的像素， $C_1$  对应于灰度值在  $[T, L-1]$  之间的像素，则  $C_0$  和  $C_1$  的概率分别为

$$w_0 = \sum_{i=0}^{T-1} p_i \quad (13-4)$$

$$w_1 = \sum_{i=T}^{L-1} p_i = 1 - w_0 \quad (13-5)$$

$C_0$  和  $C_1$  的均值分别为

$$u_0 = \sum_{i=0}^{T-1} ip_i / w_0 \quad (13-6)$$

$$u_1 = \sum_{i=T}^{L-1} ip_i / w_1 \quad (13-7)$$

整个图像的灰度均值为

$$u = w_0 u_0 + w_1 u_1 \quad (13-8)$$

定义类间方差为

$$\sigma^2 = w_0(u_0 - u)^2 + w_1(u_1 - u)^2 = w_0 w_1 (u_0 - u_1)^2 \quad (13-9)$$

令  $T$  在  $[0, L-1]$  范围内, 以步长 1 依次递增取值, 当  $\sigma^2$  取最大值时对应的  $T$  即为最佳阈值。

经二值化后的图像如图 13-5 所示。



图 13-5 二值化后的图像

### 3. 开运算

在经过二值化处理以后, 图像上仍然有一些面积较小的背景区域显示为白色, 为了去除这些噪声, 需要对图像进行开运算处理。开运算可以去除较小的明亮细节, 保持图像整体的灰度和较大的明亮区域不变。开运算的具体操作步骤是先对图像进行腐蚀运算, 然后再进行膨胀运算。先进行的腐蚀运算可以去除较小的亮细节, 但这样会使图像变暗; 接下来进行的膨胀运算又会增强图像的整体亮度, 但不会将腐蚀运算去除的部分重新引入图像中。经开运算后的图像如图 13-6 所示。



图 13-6 开运算后的图像

#### 4. 去除离散杂点噪声

经过开运算后有一些离散的小连通区域（噪声）出现在图像上，可以再次对图像进行腐蚀以去除它们，但这样很有可能会丢掉车辆自身的像素，不利于目标车辆的提取。本章的车型识别系统采用了统计连通区域面积的方法去除噪声，具体方法如下。

- 1) 提取图像中所有连通区域的外部轮廓。
- 2) 计算所有轮廓的面积。
- 3) 将所有轮廓面积一一与给定的阈值做比较，若小于给定阈值则转到步骤 4。
- 4) 从轮廓上选取一个点，将与之相邻且像素值相近的像素设置为背景色（黑色）。

经过处理以后那些小的连通区域已经被去除，效果如图 13-7 所示。

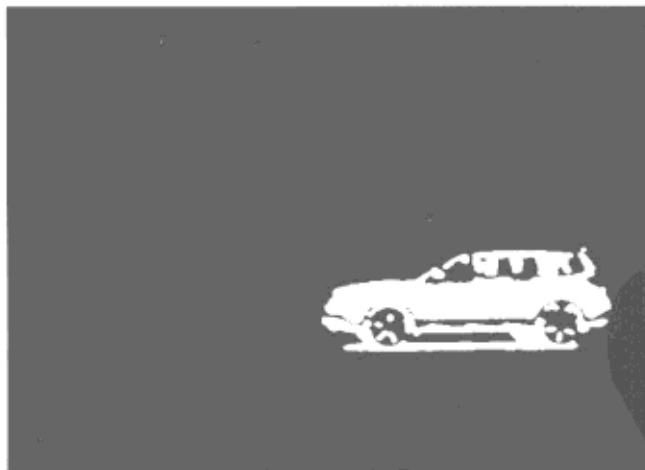


图 13-7 去噪后的图像

#### 5. 图像填充

提取到的车辆图像内部存在离散的小连通区域和空洞，且车型外部边缘也不是闭合的。为了便于提取车辆的信息特征，采用图像填充方法对图像进行闭合，图像填充分为纵向填充和横向填充，本系统采用的填充算法思想是将图像每行和每列的第一个和最后一个白色像素之间的所有像

素都填充为白色。

图像纵向填充的算法如下。

- 1) 对图像的一列从上向下进行扫描。
- 2) 如果遇到第一个白色点,记下其坐标  $y_1$ ,转到步骤 3。
- 3) 从下向上扫描,如果遇到第一个白色点,记下其坐标  $y_2$ ,转到步骤 4。
- 4) 将  $y_1$ 、 $y_2$  之间的所有点设置为白色。
- 5) 返回步骤 1 开始下一列的扫描。

图像横向填充的原理与图像纵向填充的原理相似,这里就不再详细介绍了。纵向和横向填充后的图像如图 13-8 和图 13-9 所示。



图 13-8 纵向填充后的图像



图 13-9 横向填充后的图像

### 13.1.2 车型特征提取技术

本章的车型识别系统是采用基于轮廓矩匹配技术来进行车型分类的,所以要提取的车型特征就是整辆车的轮廓,根据图像预处理阶段得到的车辆图像,我们只需提取其外部轮廓即可。外部轮廓的提取采用的是 OpenCV 里面的 `cvFindContours()` 函数,这里不做详细介绍。得到的车辆轮廓如图 13-10 所示。



图 13-10 车辆轮廓



### 13.1.3 车型分类识别技术

车型分类不但是公路自动收费系统中的要求，在停车场的车辆管理和交通流量统计等方面都有应用。目前国内外主要采用的分类方法有：感应线圈法、红外探测法、超声波检测法、车牌识别法及轮廓识别法等。

#### 1. 国内外主要采用的分类方法

##### (1) 感应线圈法

感应线圈作为车型识别的传感组件，埋设在收费车道内与挡车器的后端，通过馈线与检测系统相连。感应线圈通以高频电流后形成磁场区，当有车辆从环形感应线圈上方通过时，车体底盘的铁磁材料与环形线圈产生涡流效应，使得环形线圈电感量发生变化，由于线圈是振荡电路的一部分，因此可以检测出震荡频率的变化量及其时序，对这种反映车辆特征的频率曲线进行归类，也就实现了车型分类。

该方法将车辆底盘形状的三维信息转变为感应到的一维信号，模糊了实际的物理意义，从而加大了分析信号的难度和识别车型的难度。用感应线圈进行车型分类还存在一个问题，即车辆在线圈上方停止或改变速度时，将在很大程度上影响到识别的准确率，而这些现象又是不可避免的。

##### (2) 红外探测法

红外探测法是利用布置在车道两侧的红外数组检测器，根据汽车行驶经过时不同部位对发射装置的不同阻挡作用，采集车辆的侧面几何数据。由于该系统采用的红外检测点非常多，可以采集到大量的数据，除了车头高度、轴数、轮距及车长等特征数据外，还有其他大量信息，能够比较完整和细致地描绘出车辆的外轮廓及局部典型特征。最后通过这些数据与车型数据库的数据相比较，判断出车型，从而实现车型的自动分类。

该方法原理简单，物理概念清晰明了。但由于硬件系统较为复杂，而且系统的环境适应能力较差，故障率较高及维修不便等原因，在实际使用中难以被推广。

##### (3) 超声波检测法

超声波检测系统利用路面反射，在路面和检测单元之间没有遮挡物时，检测单元接收从路面发射的回波。系统在每次发射超声波前，根据检测到的回波信号到达时间确定本周期有无收到路面发射的回波信号，以此确定并记录在路面与检测单元之间有无遮挡物。最后由信息融合单元按融合方案和模糊识别技术确认所测量到的车辆类型。超声波检测法与上述红外探测法并无本质的区别。

##### (4) 车牌识别法

车牌识别法根据汽车牌照的图像（从摄像机获取的）识别出车牌号后，到数据库中去检索与此车牌号相对应的车型，以此做出判决。该方法对硬件要求不高，而且安装方便灵活，但需要事先对每一辆汽车创建车型牌照数据库，而且在车牌模糊的情况下不能准确识别，因此在实际应用

中受到限制。

### (5) 轮廓识别法

轮廓识别法基于摄像头获得车辆的原始图像，并从图像中分割出车辆图像，再对所得到的车辆图像进行二值化、填充、修饰和细化等一系列处理，得到一幅完整的车辆轮廓图像。然后从轮廓图像中提取车辆的几何特征，如顶棚长度与车辆长度的比值、顶棚长度与车辆高度的比值、以顶棚中垂线为界前后两部分的比值，以及车辆长度与车辆高度的比值等。轮廓识别法可以获取车辆几何特征的关键信息，并且不需要复杂的硬件设施，易于安装和维护，也不需要建立庞大的车量数据库。

综上所述，每种分类方法都有其自身的缺点和优点，相对于感应线圈和红外探测等物理检测方法，基于图像的轮廓识别法具有很大的优势。图像的信息量很大，不易造成车辆信息的丢失；硬件检测设备安装简便，只需一台摄像机即可；设备的位置便于调整；并且其更换和维护工作不会影响正常的交通。因此本章的车型识别系统采用的是轮廓识别法。

## 2. 重点介绍轮廓识别法

基于图像轮廓的车辆分类采用的方法主要分为两种：简单参数分类法和模板匹配法。

### (1) 简单参数分类法

简单参数分类法是事先对各种车辆建立一个参数数据库，这里主要是将车高、车宽、车距、顶长比（顶棚长度与车辆长度之比）、顶高比（顶棚长度与车辆高度之比）和前后比（以顶棚中垂线为界，前后两部分之比）等车辆几何尺寸作为特征参数，将车辆划分为小汽车、越野车、卡车和客车等几种类型，然后设计分类器，对车型进行分类。这种方法的优点是分类较准确，但是实用性不强，因为完整的车型很难从图像中提取出来，图片的大小不确定，车辆种类繁多，而且有时候拍摄到的车辆会有一定角度，这些都给车辆的分类识别带来了困难。

### (2) 模板匹配法

模板匹配法是事先对各种车辆建立一个模板数据库，这里主要是记录车辆的外部轮廓。然后将待分类的模板与库中的模板进行匹配，选取最相似的匹配模板作为结果输出。模板匹配法的优点是车辆特征简单、容易提取，抗干扰能力较强。

本章的车型识别系统选用的是基于轮廓矩的模板匹配法。选用轮廓的矩作为轮廓的特征有许多优点。下面简单介绍一下矩的知识。

#### 1) 矩的基本原理。

矩定义为

$$m_{p,q} = \iint x^p y^q f(x,y) dx dy \quad p, q = 0, 1, 2, \dots \quad (13-10)$$

其中， $p$  和  $q$  可取所有的非负整数值，参数  $p+q$  称为矩的阶。

通常，如下定义一个轮廓的矩：

$$m_{p,q} = \sum_{i=1}^n I(x,y)x^p y^q \quad (13-11)$$

在公式中  $p$  对应  $x$  维度上的矩,  $q$  对应  $y$  维度上的矩, 阶数表示对应的部分的指数。该计算是对轮廓边界上所有像素(数目为  $n$ ) 进行求和。如果  $p$  和  $q$  全部为 0, 那么  $m_{00}$  实际上对应轮廓边界上点的数目。

在很多实际应用中, 直接采用某阶矩来比较效果并不理想, 因为形状相同大小不同的轮廓的某阶矩会有不同的值。再者, 上述所介绍的矩依赖于所选的坐标系, 这意味着轮廓旋转后就无法正确匹配。下面介绍中心矩和归一化矩。

中心矩:

$$\mu_{p,q} = \sum_{i=0}^n I(x,y)(x-x_{avg})^p (y-y_{avg})^q \quad (13-12)$$

其中  $x_{avg} = m_{10} / m_{00}$  且  $y_{avg} = m_{01} / m_{00}$

归一化矩和中心矩也基本相同, 除了每个矩都要除以  $m_{00}$  的某个幂:

$$\eta_{p,q} = \frac{\mu_{p,q}}{m_{00}^{(p+q)/2+1}} \quad (13-13)$$

最后来介绍  $Hu$  矩,  $Hu$  矩是归一化中心矩的线性组合。之所以这样做是为了能够获取代表图像某个特征的矩函数, 这些矩函数对某些变化, 如缩放、旋转和镜像映射(除了  $h_1$ ) 具有不变性。

$Hu$  矩从中心矩中计算得到, 其计算公式如下所示:

$$\left. \begin{aligned} h_1 &= \eta_{20} + \eta_{02} \\ h_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\ h_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\ h_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\ h_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} - \eta_{03})^2] \\ &\quad + (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{21} + \eta_{03})^2 - (\eta_{21} - \eta_{03})^2] \\ h_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] + 4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\ h_7 &= (3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\ &\quad - (\eta_{30} - \eta_{12})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \end{aligned} \right\} \quad (13-14)$$

## 2) 矩的函数实现。

系统采用的是 OpenCV 中的矩匹配函数, 其函数原型如下:

```
double cvMatchShapes(const void* object1,
                     const void* object2,
```

```
int method,  
double parameter=0);
```

下面主要介绍一下函数中的第三个参数 method，见表 13-1。

表 13-1 cvMatchShapes()使用的匹配方法

method 取值	cvMatchShapes()返回值
CV_CONTOURS_MATCH_11	$I_1(A, B) = \sum_{i=1}^7 \left  \frac{1}{m_i^A} - \frac{1}{m_i^B} \right $
CV_CONTOURS_MATCH_12	$I_2(A, B) = \sum_{i=1}^7  m_i^A - m_i^B $
CV_CONTOURS_MATCH_13	$I_3(A, B) = \sum_{i=1}^7 \left  \frac{m_i^A - m_i^B}{m_i^A} \right $

在表中  $m_i^A$  和  $m_i^B$  被定义为

$$m_i^A = \text{sign}(h_i^A) \cdot \log |h_i^A| \quad (13-15)$$

$$m_i^B = \text{sign}(h_i^B) \cdot \log |h_i^B| \quad (13-16)$$

在这里  $h_i^A$  和  $h_i^B$  分别是  $A$  和  $B$  的  $Hu$  矩。

## 13.2 系统功能

车型识别系统的主要功能是采用基于轮廓匹配的方法对车型进行分类识别。

### 13.2.1 功能描述

车型识别系统具体功能如下：

- 1) 读入 BMP 格式的背景和前景图像。
- 2) 通过背景消除法去除大部分的背景。
- 3) 对图像进行开运算进一步去除噪声。
- 4) 统计图像中各个连通分量的面积，完全清除噪声。
- 5) 提取车辆的轮廓。
- 6) 根据提取的车辆轮廓对车辆进行分类。

### 13.2.2 界面效果

车型识别系统的界面效果如图 13-11 所示。



图 13-11 车型识别系统界面效果

## 13.3 系统结构与流程

车型识别系统从最初读入图像到最终输出分类结果，大致可分为 5 个主要模块，各模块之间是顺序结构。

### 13.3.1 总体结构

车型识别系统的总体结构如图 13-12 所示。

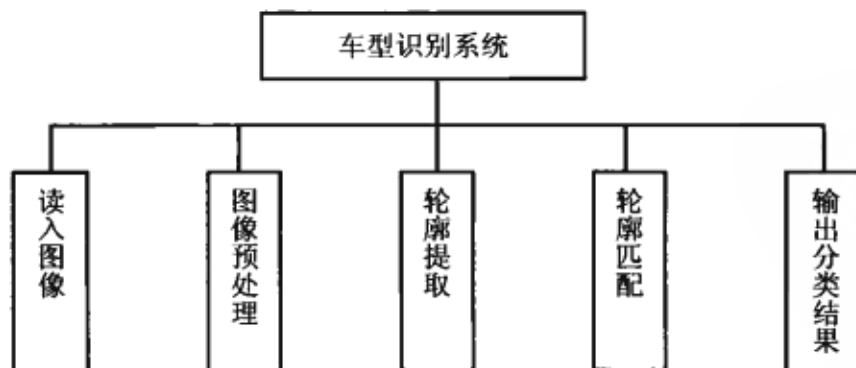


图 13-12 车型识别系统总体结构图

### 13.3.2 主要流程

车型识别系统的主要流程如图 13-13 所示，纵向填充算法的流程如图 13-14 所示。

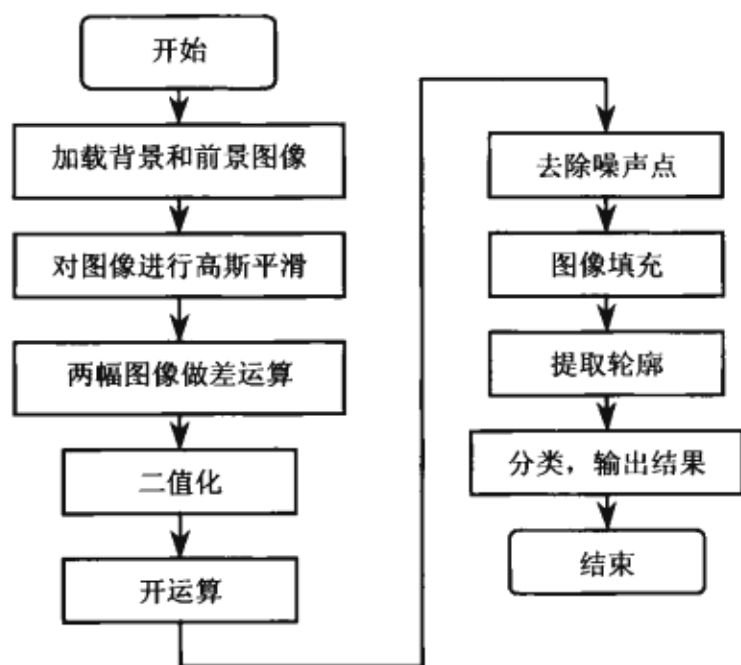


图 13-13 车型识别系统流程图

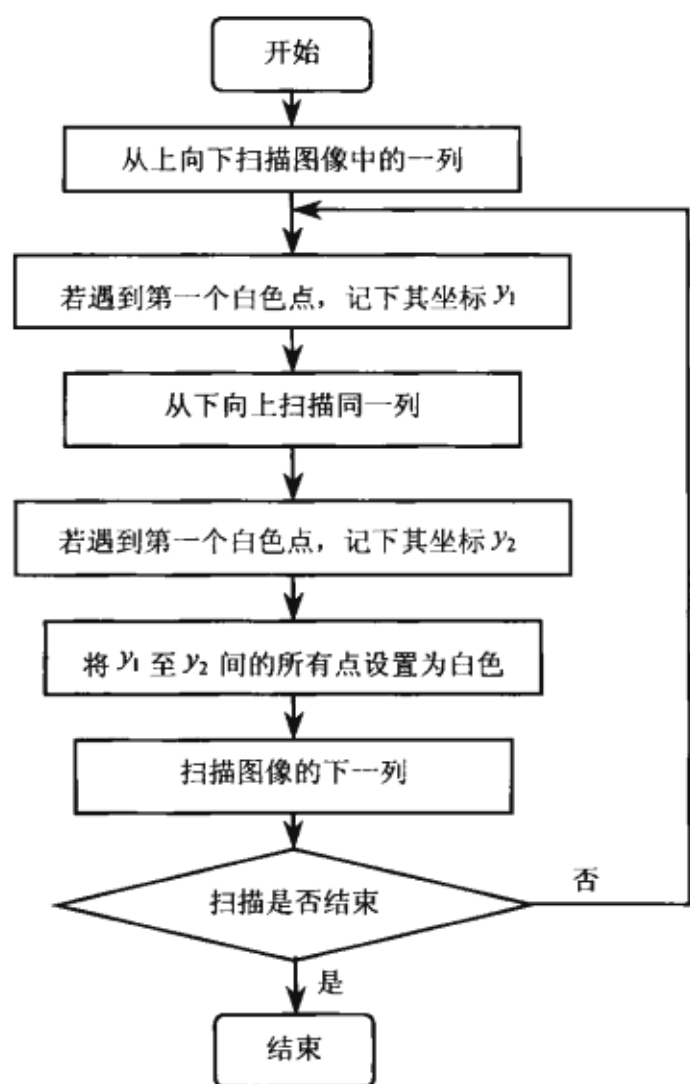


图 13-14 纵向填充算法的流程图

## 13.4 编程实现

车型识别系统采用 Visual Studio 2008 开发平台结合 OpenCV 函数库编程实现。

### 13.4.1 变量定义模块

车型识别系统用到的变量定义如下：

```
IplImage *BkImg;           //背景图像
IplImage *FrImg;           //前景图像
IplImage *BkGray;         //背景灰度图像
IplImage *FrGray;         //前景灰度图像
IplImage * SubImg;        //做差后的图像
IplImage * BinaryImg;     //二值图像
CvSeq * contour;          //存放轮廓的序列
myimg m_Cimage;           //显示图像
```



### 13.4.2 图像显示模块

图像显示模块自定义了 myimg 图像类，用于载入并显示位图图像。

```
#pragma once
#include "highgui.h"
#include "cv.h"
class myimg:public CvvImage
{
public:
    myimg();
    void mSetImg(IplImage *pImg);
public:
    ~myimg();
}
;
//以下是 myimg 类的实现
myimg::myimg()
{
}
void myimg::mSetImg(IplImage * pImg)
{
    m_img=cvCloneImage(pImg);
}
myimg::~myimg(void)
{
}

//以下是图像的显示部分
void CCarShapeIdentifyView::OnDraw(CDC* pDC)
{
    CCarShapeIdentifyDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;
    // TODO: 此处为本机数据添加绘制代码
    myimg & img=pDoc->m_Cimage;
    CRect m_rect;
    GetClientRect(&m_rect);
    //计算图像的宽度和高度，能够按照原始大小显示
    m_rect.right=img.Width();
    m_rect.bottom=img.Height();
    img.DrawToHDC(pDC->GetSafeHdc(),m_rect);
}
```

### 13.4.3 载入图像模块

由于所实现的车型识别系统是基于 MFC 的单文档应用程序, 因此文件的打开模块采用的是基于 Windows 的对话框模式。以下是载入背景图像的代码实现部分, 由于载入前景图像也采用了这种方法, 这里便不再列出其代码。

```
void CCarShapeIdentifyDoc::OnLoadbk()
{
    // TODO: 在此添加命令处理程序代码
    LPCTSTR lpszFilter="BMP Files (*.bmp)|*.bmp|任何文件|*.*||";
    CFileDialog dlg1(TRUE,lpszFilter,NULL,OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
        lpszFilter,NULL);
    //打开文件对话框
    if(dlg1.DoModal()==IDOK)
    {
        if(0==(BkImg=cvLoadImage(dlg1.GetPathName(),1)))
        {
            AfxMessageBox("无法打开文件!");
            return;
        }
        else
        {
            m_Cimage.mSetImg(BkImg);
            UpdateAllViews(NULL);
            return ;
        }
    }
}
```

### 13.4.4 车辆提取模块

车辆提取模块包含图像处理常用的一些方法, 这些方法都很基本, 故这里不再介绍, 只将代码列出, 必要的解释通过程序注释给出。

```
//图像做差
void CCarShapeIdentifyDoc::OnImgsub()
{
    //将图像转化为灰度图像
    BkGray=cvCreateImage(cvGetSize(BkImg),8,1);
    cvCvtColor(BkImg,BkGray,CV_BGR2GRAY);
    FrGray=cvCreateImage(cvGetSize(BkImg),8,1);
    cvCvtColor(FrImg,FrGray,CV_BGR2GRAY);
    //对图像进行高斯平滑操作
    cvSmooth(BkGray,BkGray,CV_GAUSSIAN,5);
    cvSmooth(FrGray,FrGray,CV_GAUSSIAN,5);
    //对图像进行做差运算
```

```

        SubImg=cvCreateImage(cvGetSize(BkImg),8,1);
        cvAbsDiff(FrGray,BkGray,SubImg);
        //显示图像
        m_Cimage.mSetImg(SubImg);
        UpdateAllViews(NULL);
    }

    //二值化
    void CCarShapeIdentifyDoc::OnBinary()
    {
        //图像二值化
        BinaryImg=cvCreateImage(cvGetSize(BkImg),8,1);
        cvThreshold(SubImg,BinaryImg,0,255,CV_THRESH_OTSU);
        //显示图像
        m_Cimage.mSetImg(BinaryImg);
        UpdateAllViews(NULL);
    }

    //开运算
    void CCarShapeIdentifyDoc::OnOpen()
    {
        //定义腐蚀和膨胀用的核
        IplConvKernel * kernel_5_5;
        kernel_5_5=cvCreateStructuringElementEx(5,5,2,2,CV_SHAPE_ELLIPSE,0);
        IplConvKernel * kernel_7_7;
        kernel_7_7=cvCreateStructuringElementEx(7,7,3,3,CV_SHAPE_ELLIPSE,0);
        //对前景图像进行开闭合运算以除去杂点,分割出运动物体
        //对图像进行腐蚀运算
        cvErode(BinaryImg,BinaryImg,kernel_5_5,1);
        //对图像进行膨胀运算
        cvDilate(BinaryImg,BinaryImg,kernel_7_7,1);
        //显示图像
        m_Cimage.mSetImg(BinaryImg);
        UpdateAllViews(NULL);
    }

    //去除噪声
    void CCarShapeIdentifyDoc::OnRemovenoise()
    {
        //去除较小的离散连通区域
        //首先将图像复制一份
        IplImage * BinaryImg2;
        BinaryImg2=cvCloneImage(BinaryImg);
        //定义内存空间
        CvMemStorage * storage=cvCreateMemStorage(0);
        //定义统计联通区域轮廓的序列
        CvSeq * area_contour=NULL;

```

## 数字图像处理典型案例详解

```

int Number_contours=0;    //轮廓个数
double Area=0;           //轮廓面积
CvScalar s_fill_new;
s_fill_new.val[0]=0;
CvScalar loDiff;
loDiff.val[0]=10;
CvScalar upDiff;
upDiff.val[0]=10;
//查找图像中的最外部轮廓
Number_contours=cvFindContours(BinaryImg,storage,&area_contour,
                                sizeof(CvContour),CV_RETR_EXTERNAL);
//统计轮廓面积,对于面积小于阈值的连通区域进行删除
for(CvSeq * c=area_contour;c!=NULL;c=c->h_next)
{
    //获得轮廓面积
    Area=fabs(cvContourPerimeter(c));
    if(Area<30)
    {
        for(int i=0;i<1;i++)
        {
            //取轮廓上的一个点作为图像填充种子点
            CvPoint * seedpoint=CV_GET_SEQ_ELEM(CvPoint,c,i);
            //用漫水填充算法将小的轮廓填充为背景色
            cvFloodFill(BinaryImg2,*seedpoint,s_fill_new,loDiff,upDiff, NULL,4,0);
        }
    }
}
BinaryImg=cvCloneImage(BinaryImg2);
//图像显示
m_Cimage.mSetImg(BinaryImg);
UpdateAllViews(NULL);
}

//图像填充
void CCarShapeIdentifyDoc::OnImgfill()
{
    //对汽车进行内部填充
    CvScalar s;
    CvScalar s_new;
    s_new.val[0]=255;
    //记录填充行的最左位置
    int left_X=0;
    //记录填充行的最右位置
    int right_X=0;
    //记录填充列的最上位置
    int top_Y=0;
    //记录填充列的最下位置

```

```
int bottom_Y=0;
//纵向填充
for(int wt=0;wt<BinaryImg->width;wt++)
{
    for(int ht=0;ht<BinaryImg->height;ht++)
    {
        s=cvGet2D(BinaryImg,ht,wt);
        if(255==s.val[0])
        {
            top_Y=ht;//找到最上边的白色点
            for(int m=BinaryImg->height-1;m>=0;m--)
            {
                s=cvGet2D(BinaryImg,m,wt);
                if(255==s.val[0])
                {
                    bottom_Y=m;
                    break; //跳出寻找最下边点的循环
                }
            }
            //将 top_Y 与 bottom_Y 之间的点设置为白色
            for(int i=top_Y;i<=bottom_Y;i++)
                cvSet2D(BinaryImg,i,wt,s_new);
            ht=BinaryImg->height;//强制跳出此列循环
        }
    }
}
//横向填充
for(int ht=0;ht<BinaryImg->height;ht++)
{
    for(int wt=0;wt<BinaryImg->width;wt++)
    {
        s=cvGet2D(BinaryImg,ht,wt);
        if(255==s.val[0])
        {
            left_X=wt;//找到最左边的白色点
            for(int m=BinaryImg->width-1;m>=0;m--)
            {
                s=cvGet2D(BinaryImg,ht,m);
                if(255==s.val[0])
                {
                    right_X=m;
                    break; //跳出寻找最右边点的循环
                }
            }
            //将 left_X 与 right_X 之间的点设置为白色
            for(int i=left_X;i<=right_X;i++)
                cvSet2D(BinaryImg,ht,i,s_new);
        }
    }
}
```

```

        wt=BinaryImg->width;//强制跳出此行循环
    }
}
//显示图像
m_Cimage.mSetImg(BinaryImg);
UpdateAllViews(NULL);
}

```

### 13.4.5 轮廓提取模块

车型识别系统采用的是轮廓匹配方法，提取轮廓至关重要，这里采用的是 OpenCV 函数库里的函数，轮廓提取代码如下：

```

void CCarShapeIdentifyDoc::OnContour()
{
    //轮廓提取
    //创建内存空间
    CvMemStorage * storage2=cvCreateMemStorage(0);
    //提取轮廓
    cvFindContours(BinaryImg,storage2,&contour,sizeof(CvContour),CV_RETR_EXTERNAL);
    //将轮廓绘制到图像 blackImg 上
    IplImage* blackImg;
    blackImg=cvCreateImage(cvGetSize(BinaryImg),8,1);
    cvZero(blackImg);
    if(contour)
        cvDrawContours(blackImg,contour,cvScalarAll(255),cvScalarAll(255),100);
    //显示图像
    m_Cimage.mSetImg(blackImg);
    UpdateAllViews(NULL);
}

```

### 13.4.6 车型识别模块

车型是通过轮廓匹配法进行识别的，即将提取出的车辆轮廓与事先提取的轮廓模板进行匹配，选取最匹配的车型轮廓作为识别结果，其代码如下：

```

void CCarShapeIdentifyDoc::OnIdentify()
{
    //进行轮廓的匹配
    //载入模板
    IplImage *sample[4];
    sample[0]=cvLoadImage("轮廓\\0.bmp",0);
    sample[1]=cvLoadImage("轮廓\\1.bmp",0);
    sample[2]=cvLoadImage("轮廓\\2.bmp",0);
    sample[3]=cvLoadImage("轮廓\\3.bmp",0);
    //提取模板轮廓

```



```

CvMemStorage *storage_sample=NULL;
storage_sample=cvCreateMemStorage(0);
cvClearMemStorage(storage_sample);
CvSeq* contours_data[4];
for(int i=0;i<4;i++)
    contours_data[i]=0;
for(int i=0;i<4;i++)
    cvFindContours(sample[i],storage_sample,&contours_data[i],
        sizeof (CvContour),
CV_RETR_EXTERNAL);
//进行轮廓匹配
double xiangshidu[4];//匹配系数数组
memset(xiangshidu,0,sizeof(xiangshidu));
for(int i=0;i<4;i++)
    xiangshidu[i]=cvMatchShapes(contour,contours_data[i],2);
//查找相似度最小的值,即为最匹配的车型
int BestMatch=0;
double min_xiangshidu=xiangshidu[0];//将第一个匹配值假设为最小值
for(int i=0;i<4;i++)
{
    if(xiangshidu[i]<min_xiangshidu)
    {
        min_xiangshidu=xiangshidu[i];
        BestMatch=i;
    }
}
//输出分类结果
switch (BestMatch)
{
    case 0:
        AfxMessageBox("车型为: 小汽车");
        break;
    case 1:
        AfxMessageBox("车型为: SUV");
        break;
    case 2:
        AfxMessageBox("车型为: 商务车");
        break;

    case 3:
        AfxMessageBox("车型为: 面包车");
        break;
    default:
        AfxMessageBox("识别失败");
        break;
}
}

```

## 13.5 经验分享

在实际的车型识别系统开发过程中，以下问题需要认真研究处理，以免影响开发效率和识别效果。

1) 目标车辆分割问题。分割目标车辆在整个系统中的地位非常重要，如果分割不准确，后面的处理过程及识别结果就会不准确。目标车辆分割的方法很多，但每种方法都有它的局限性，要根据实际应用场合认真研究使用哪种分割方法，尤其要考虑到阴影和行人的影响。

2) 车型分类识别问题。按照收费标准规定的同一档车型中，不同品牌、型号的车辆外形都不是完全相同的，甚至差异很大（如同是小轿车，两厢车和三厢车的外形差异就很大），因此，车型分类方法要有一定的泛化能力，以适应同一类车型的不同车辆外形之间的差别。当然，可以对车辆外形进行精细的分类识别，然后根据事先建立好的模型库来判定这个“精细类别”对应收费标准中的哪类车型，但这样无疑会增加识别程序的复杂度。

3) 程序的封装与接口问题。在实际应用中，从数据读取到结果输出之间的全部中间处理过程应该都是自动完成的，处理细节被完全封装起来，对用户是透明的。输入的数据也不一定是图像文件，有时是直接从采集卡中读取的视频数据。输出的结果也不再是弹出简单的对话框，而是直接存入收费管理系统的数据库。

## 第 14 章 车牌识别系统

古时候跑江湖的都讲究“人过留名，雁过留声”，人不留名不知张王李赵，雁不留声不知春夏秋冬。那现在一辆汽车驶过智能交通系统的监控镜头后会留下什么呢？除了车型靓影，可能就是车牌号码了。法国是最早使用汽车牌照的国家，1892 年 8 月 14 日颁布的《巴黎警察条例》规定，所有汽车都必须挂上印有车主姓名、住址以及登记号码的金属牌照。汽车号牌是国家车辆管理法规规定的具有统一式样的带有注册登记编码的号码牌，是识别车辆身份的标识。车牌识别技术在交通监控、公路收费、停车收费、汽车防盗、违章管理等方面均有重要应用。车牌识别系统是智能交通系统中不可或缺的核心组成部分。本章解读车牌识别的基本原理及车牌识别系统的编程实现方法。

**本章要点：**

- 车牌图像预处理技术
- 车牌定位技术
- 车牌字符分割技术
- 车牌字符识别技术
- 车牌识别系统功能描述
- 车牌识别系统的总体结构与主要流程
- 车牌识别系统的编程实现

### 14.1 核心技术原理

车牌识别系统涉及的核心技术主要包括车牌图像预处理、车牌定位、字符分割和字符识别技术。前 3 项技术属于数字图像处理技术范畴，而字符识别则属于模式识别技术范畴。

#### 14.1.1 车牌图像预处理技术

车牌图像的采集是车牌识别的第一步，只有采集到合适的图像并进行适当的处理，才能够进行后续的字符分割与识别工作。由于图像拍摄环境及车牌自身的影响（如光线不均匀、噪声干扰、

角度不正及车牌受到污染等因素),使得采集到的原始图像不能满足后续处理的质量要求,这时就需要对采集到的图像进行识别前的预处理,突出图像中感兴趣的部分,以便准确定位车牌和分割车牌字符,进而提高车牌号码的识别率。

下面以图 14-1 所示图像为例来介绍图像预处理的过程和方法。

### 1. 灰度化

一般通过摄像头或数码照相机采集图像,或者是计算机中存储的图像通常是彩色图像,即 24 位真彩色图像。彩色图像包含着大量的色彩信息,不仅增大了内存的开销也大大降低了处理速度,因此通过将彩色图像灰度化来加快处理速度。图 14-2 是颜色转化后的灰度图像。



图 14-1 采集到的原始图像



图 14-2 车辆灰度图像

### 2. 灰度拉伸

有时因为光线问题会造成图像局部过亮或过暗,这就需要对图像进行拉伸使之覆盖较大的取值区间。使得亮的区域更亮,暗的区域更暗,提高图像的对比度,从而使图像边缘明显,使车牌的信息更加清楚地表现出来。灰度拉伸是将灰度图像进行分段性变化,若原图像  $f(x,y)$  灰度变化区间为  $[a,b]$ ,希望变换后图像  $g(x,y)$  的灰度范围扩展到区间  $[c,d]$ ,可采用下述线性变换来实现。

$$g(x,y) = [(d-c)/(b-a)]f(x,y) + c \quad (14-1)$$

其中  $g(x,y)$  是变换后的图像的像素值,  $f(x,y)$  是原图像的像素值。

通常一幅图像中大部分像素的灰度分布在  $[a,b]$  之间,小部分像素的灰度级强度超出此区间。为改善拉伸效果,可令:

$$g(x,y) = \begin{cases} c & 0 \leq f(x,y) < a \\ \left(\frac{d-c}{b-a}\right)f(x,y) + c & a \leq f(x,y) \leq b \\ d & b < f(x,y) \leq 255 \end{cases} \quad (14-2)$$

灰度变换原理如图 14-3 所示。

灰度拉伸后的图像如图 14-4 所示。从图中可以看出整个图像的对比度明显加强,特别是车牌区域。

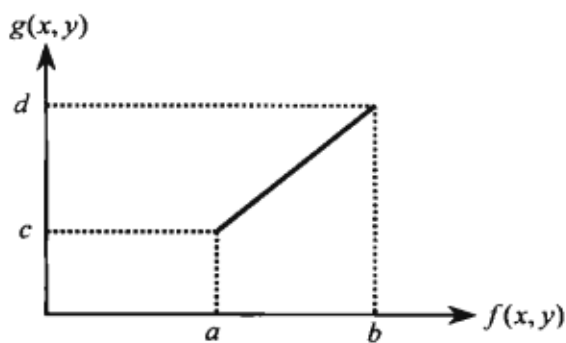


图 14-3 灰度变换原理图

### 3. 图像腐蚀

腐蚀的作用是消除物体边界点，是边界向内部收缩的过程，腐蚀后的图像如图 14-5 所示。从图中可以看出字符的边缘明显向内部收缩。



图 14-4 灰度拉伸后的图像



图 14-5 腐蚀后的图像

### 4. 原始图像与腐蚀后的图像相减

由于腐蚀后的图像中字符的边缘向内收缩，所以用原图像减去腐蚀后的图像可以得到车牌的总体轮廓。相减后的图像如图 14-6 所示。从图中可以看到车牌字符的清晰轮廓。

### 5. 二值化

二值图像是指整幅图像内仅有黑和白两个值。为了加快处理速度并能够将车牌字符与车牌背景分开，通过阈值的设定将灰度值小于阈值的像素直接设置为 0，灰度值大于阈值的像素直接设置为 255。阈值的选择至关重要，系统采用的是最大方差阈值分割法，也叫 Otsu 法。其具体原理请参看第 13 章。二值化后的图像如图 14-7 所示。



图 14-6 相减后的图像



图 14-7 二值化后的图像

## 6. 中值滤波

考虑到文字是由许多短竖线组成的, 而背景噪声大部分是孤立的, 故用模板  $(1,1,1,1)^T$  对二值图像进行中值滤波, 以去除孤立噪声。滤波后的图像如图 14-8 所示。

## 7. 开运算

滤波后的图像存在一些较大的区域噪声, 为了去除这些噪声, 对图像进行开运算操作。使用同一个结构元素对目标图像先进行腐蚀运算, 再进行膨胀运算的过程称为开运算。原图经过开运算后, 能够去除孤立的小点、毛刺和小的连通区域, 去除小物体、平滑较大物体的边界, 同时并不明显改变其面积。开运算后的图像如图 14-9 所示, 经过开运算后很多噪声都被除去了。



图 14-8 滤波后的图像



图 14-9 开运算后的图像

### 14.1.2 车牌定位技术

车牌定位是对预处理后的图像中的车牌位置进行准确的定位。车牌定位在整个车牌识别系统中至关重要, 只有准确的定位车牌才能为后续的识别工作提供有力的保障。目前车牌的定位有很多方法, 本章采用水平投影和垂直投影来定位出车牌的位置。下面介绍车牌定位的过程。

#### 1. 垂直投影

从图像中可以看出车牌区域的较亮的像素值比较多且比较集中, 故将图像做垂直投影, 通常车牌靠近整幅图像的下部, 因此由下向上扫描。对前面处理过的图像的像素沿着垂直方向累加产生一个车牌图像的投影分布, 车牌位置应对应投影分布的某段像素累加和大小均匀的波段。投影后的大致效果如图 14-10 所示。

为了能够准确地定位车牌的高度位置, 对投影的累加和进行聚类, 具体的思想是, 按照图像投影时的扫描顺序, 如果某一行的像素累加和除以上一行像素的累加和小于一个给定的阈值, 则将其像素累加和置零。统计归类后取连续不为零的累加和的宽度即可, 选择和车牌高度最为接近的那段连续区域的开始和结

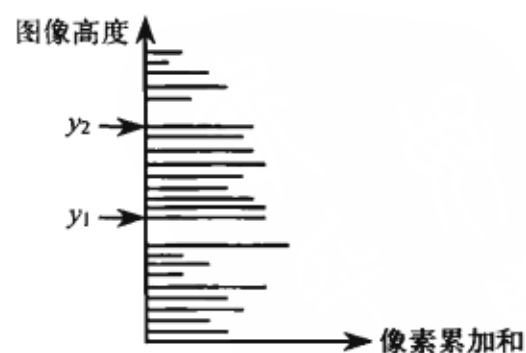


图 14-10 垂直投影效果图



束坐标,此即为车牌的高度坐标(图 14-10 中的  $y_1$  与  $y_2$ ),考虑到图像尺寸的不同,本章直接选取了最长连续区域作为车牌的高度位置。投影后切割出的车牌高度区域如图 14-11 所示。

## 2. 水平投影

水平投影和垂直投影相似。这里不再做详细的介绍。由于车牌的倾斜,切割出来的车牌图像的边缘部分损失了一些内容,故在最后定位时适当向外延伸一些像素。最后得到的车牌图像如图 14-12 所示。



图 14-11 车牌高度图像



图 14-12 投影后得到的车牌图像

### 14.1.3 车牌字符分割技术

切割出车牌区域后,下一步就是切割出字符,但切割出的车牌图像仍是灰度图像,故在进行字符分割之前我们需先对车牌图像进行一些处理。具体过程如图 14-13 所示。

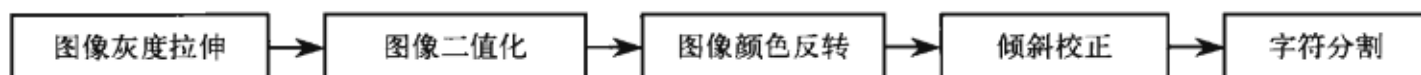


图 14-13 车牌字符分割过程

#### 1. 图像灰度拉伸

由于前面已经介绍过图像的灰度拉伸,这里不再赘述。经过拉伸后的车牌图像如图 14-14 所示。

#### 2. 图像二值化

车牌图像的二值化采用的也是最大方差阈值分割法。二值化后的图像如图 14-15 所示。

#### 3. 图像颜色反转

为了方便日后处理,在进行字符分割之前将二值化后的车牌图像颜色进行反转,即白色变成黑色,黑色变成白色。反转后的图像如图 14-16 所示。



图 14-14 灰度拉伸后的图像



图 14-15 二值化后的车牌图像



图 14-16 颜色反转后的车牌图像

#### 4. 倾斜校正

由于图像在拍摄的过程中车牌的位置可能是倾斜的,所以在字符分割之前先对车牌字符进行倾斜校正,使得字符都处于同一水平位置,这样既利于字符的分割又可以提高字符识别的准确率。调整的算法主要是根据图像上左右两边黑色像素的平均加权高度来进行的。一般来说,一张由众

多字符组成的图像，它左右两边的字符像素的高度应该处于水平位置的附近，如果两边字符像素的平均位置有比较大的起落，那就说明图像存在倾斜，需要进行调整。具体来说，首先分别计算图像左半边和右半边的像素的加权平均高度，然后求出斜率，根据斜率重新组织图像。里面包含了一个从新图像到旧图像的映射。如果新图像中的像素映射到旧图像中时超出了旧图像的范围，则把新图像中的该像素设置为白色。经过倾斜校正的车牌图像如图 14-17 所示。



图 14-17 倾斜校正后的车牌图像

### 5. 字符分割

一幅定位准确的车牌图像通常由 1 个汉字和 6 个字母或数字组成。识别时只能根据每个字符的特征来进行判断，所以要想成功地识别出车牌，还要进行字符的分割工作。

具体算法如下：

1) 先自上向下对图像进行逐行扫描，直至遇到第一个黑色的像素点，记录它。然后再自下向上对图像进行逐行扫描，直至找到第一个黑色像素点，记录下来。这样就找到了图像大致的高度范围。

2) 在这个高度范围之内再自左向右逐列进行扫描，遇到第一个黑色像素点时认为是字符分割的起始位置，然后继续扫描，直至遇到有一列中没有黑色像素点，则认为这个字符分割结束。然后继续扫描，按照上述的方法一直扫描至图像的最右端。这样就得到了每个字符的比较精确的宽度范围。

3) 去除车牌中间的小圆点。由于切割时将车牌中间的小圆点一并切割了出来，通过比较切割出来的字符图像的宽度除以车牌图像整个宽度是否大于一个给定的阈值（系统选用 0.1）来去除小圆点，若小于 0.1 则认为分割的字符是噪声并删除。

4) 由于通过上述步骤分割出的字符的高度是整个车牌图像中字符的最高和最低坐标值，所以并不准确。对切割出来的字符要进行再次的高度定位，得到每个字符的准确高度范围，方法和第 1 步的方法一样。

字符分割后的图像如图 14-18 所示，为了展示分割出来的效果，给每个字符加上边框。



图 14-18 分割出的字符图像

### 6. 字符归一化

这里的归一化是指将分割好的字符图像通过系数变换得到高度和宽度均相等的图像。因为在切割出来的图像中，字符大小存在较大的差异相对而言，同一尺寸的字符便于特征提取，识别的标准性更强，准确率自然也更高。为了和字符库中的模板匹配，再次对字符进行反色处理。字符归一化后的图像如图 14-19 所示。

### 7. 字符细化

由于归一化后的字符图像较小，而字符的笔画较大，对于字母和数字还比较好识别，而对于

汉字来说其笔画较稠密,不便于字符的识别,故在对字符识别前先进行细化。图像细化就是把二值图像中具有一定宽度的线条状区域变成一条薄线(即只有一个像素宽度)。图像细化大大压缩了原始图像的数据量,并保持其形状的基本拓扑结构不变,从而为字符识别中的特征抽取等应用奠定了基础。字符细化后的效果如图 14-20 所示。



图 14-19 归一化后的字符图像



图 14-20 细化后的字符图像

#### 14.1.4 车牌字符识别技术

车牌字符识别用到的技术已超出数字图像处理的范畴,属模式识别领域。本章直接对分割后的字符通过特征的提取进行模式识别和分类,从而识别出图像中的字符,车牌字符识别过程如图 14-21 所示。

##### 1. 字符特征提取

经过上面一系列的变换,原来大小不同,分布不规则的各个字符变成了一个个大小相同的字符。下面就要从归一化处理完毕的字符中提取最能体现这个字符特点的特征向量,将提取出的特征向量送入到神经网络中进行训练和识别。特征提取建立在需要识别的物体被分割出来的基础上,提取所需要的特征,并对某些参数进行计算和测量,根据测量结果进行分类。

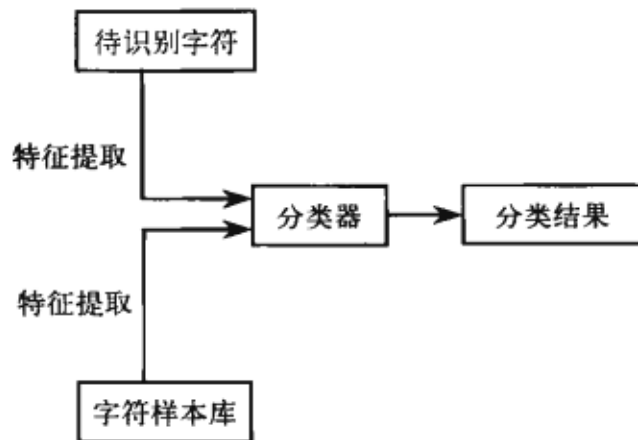


图 14-21 车牌字符识别过程

求出一组对分类有效的特征,在特征维数减少到同等水平时其分类性能最佳。这里存在一个映射关系使得新样本描述维数比原维数低。

$$A:Y \rightarrow X \quad (14-3)$$

其中每个分量  $x_i$  是原特征向量各分量的函数,即

$$x_i = x_i(y_1, y_2, \dots, y_D) \quad (14-4)$$

字符特征向量的提取方法多种多样,常用的有逐像素特征提取法、骨架特征提取法、垂直方向数据统计特征提取法、13 点特征提取法及弧度梯度特征提取法等。本章采用逐像素特征提取法。逐像素提取法将对图像进行逐行逐列扫描,当遇到黑点时取“1”或者“0”,当遇到亮点时取“0”或者“1”。扫描结束后将得到一个与图像像素个数相同的特征向量矩阵。将所提取的特征送到分类器的输入端进行分类得出结果。本章将黑点标记为“0”、亮点标记为“1”。

为了达到一个更好的识别效果,本章介绍的车牌字符识别采用了两种统计特征。一种是网格

特征, 由于模板图像都是  $20 \times 40$  大小的, 所以将待识别的字符也归一化为模板图像大小, 然后把图像分成  $5 \times 5 = 25$  个小格子, 统计每个格子里白色像素的个数, 形成一个 25 维的矢量; 另外一种特征是交叉点, 在水平方向以及垂直方向五等分的地方, 做水平线或垂直线穿过数字, 看其与数字相交的次数, 这样又获得了 8 个数值, 加起来一共是 33 维。

## 2. 分类器设计

分类器能够根据识别对象所呈现的特征实现自动分类。在介绍分类器之前要介绍 3 个概念。

- 训练集: 一个已知样本集, 用它来开发模式分类器。
- 测试集: 在设计分类系统时没有用过的独立样本集。
- 学习与训练: 从训练样本提供的数据中找出分类器设计准则函数的最优解。

### (1) 分类器设计准则

- 建立特征空间中的训练集, 已知训练集里每个点的所属类别。
- 从训练集出发, 寻求某种判别函数或判别准则, 设计判决函数模块。
- 根据训练集中的样本确定模型中的参数。
- 将这一模型用于判别, 利用判别函数或判别准则去判断每个未知类别的点应该属于哪个类。

### (2) 分类器设计基本方法

分类器设计有两种基本方法: 模板匹配法和判别函数法。

#### • 模板匹配法

将待分类样本与标准模板进行比较, 看与哪个模板匹配程度更相似, 从而确定待测样本的分类。

#### • 判别函数法

基于判别函数的分类器有两种: 基于概率统计分类和几何分类。基于概率统计的分类是指通过训练样本的概率分布进行判别函数设计, 然后用它进行分类。几何分类是把特征空间分解为对应于不同类别的子空间, 而且呈线性的分离函数, 将使计算简化。

本章介绍的字符识别采用模板匹配法, 事先采集好各个车牌字符图像, 建立一个车牌字符图像样本库。当要识别字符时, 将模板特征提取出来, 保存在文件里, 然后提取待识别字符的特征矢量, 求出其和各个模板特征矢量的加权距离, 最小的一个即为识别结果, 由于汉字笔画的繁多, 且汉字只出现在车牌字符的首位, 故将汉字和其他字符分开处理, 即汉字只和汉字样本库中的字符匹配。

## 14.2 系统功能

车牌识别系统的主要功能是对汽车牌照进行检测定位并识别出车牌号码。

### 14.2.1 功能描述

本章介绍的车牌识别系统处理的是 BMP 格式的图像文件，所以在使用时一定要保证待识别的图像为 BMP 格式。系统从最初的图像读入到最后识别出车牌主要实现了以下几点功能：

- 1) 图像灰度化及灰度拉伸。
- 2) 图像二值化。
- 3) 车牌定位与切割。
- 4) 车牌倾斜校正。
- 5) 字符分割。
- 6) 字符归一化及细化。
- 7) 字符特征提取及识别。

### 14.2.2 界面效果

车牌识别系统运行时的初始界面如图 14-22 所示。

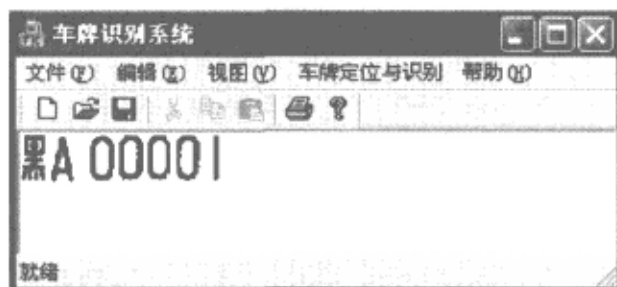


图 14-22 车牌识别系统界面效果

## 14.3 系统结构与流程

车牌识别系统从最初读入图像到最终输出分类结果，大致可分为七个主要模块，各模块之间是顺序的结构。由于系统所用到的算法较多，下面主要列出系统的总体结构和神经网络算法的流程。

### 14.3.1 总体结构

车牌识别系统的总体结构如图 14-23 所示。

### 14.3.2 主要流程

车牌识别系统的流程如图 14-24 所示。

## 14.4 编程实现

车牌识别系统采用 Visual Studio 2008 开发平台结合 OpenCV 函数库编程实现。

### 14.4.1 自定义函数模块

由于 OpenCV 库函数中参数类型和个数的限制，有时并不能达到我们想要的处理结果。在此单独列出程序中使用的灰度拉伸和中值滤波这两个函数。其中，中值滤波函数的核为  $(1,1,1,1,1)^T$ ，

读者可以根据自己的需要进行修改。

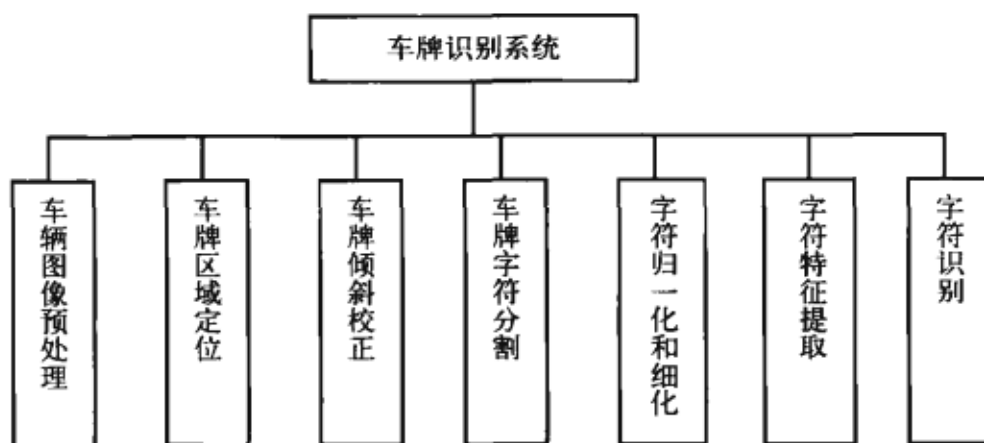


图 14-23 车牌识别系统结构图



图 14-24 车牌识别系统流程图

### 1. 灰度拉伸函数

```
void Stretch(IplImage *src, IplImage *dst, int min, int max)
{
    int low_value=min; //拉伸后像素的最小值
    int high_value=max; //拉伸后像素的最大值
    float rate=0; //图像的拉伸率
    float stretch_p[256], stretch_pl[256], stretch_num[256];
    //清空三个数组
    memset(stretch_p, 0, sizeof(stretch_p));
    memset(stretch_pl, 0, sizeof(stretch_pl));
    memset(stretch_num, 0, sizeof(stretch_num));
    //求图像各个灰度级出现的次数
    for(int y=0; y<src->height; y++)
    {
        uchar * ptr=(uchar*)(src->imageData + y*src->widthStep);
        for(int x=0; x<src->width; x++)
        {
            stretch_num[ptr[x]]++;
        }
    }
}
```



```

    }
    //求图像各个灰度级的出现概率
    for(int i=0;i<256;i++)
    {
        stretch_p[i]=stretch_num[i]/(src->width*src->height);
    }
    //求各个灰度级的概率和
    for(int i=0;i<256;i++)
    {
        for(int k=0;k<=i;k++)
            stretch_pl[i]+=stretch_p[k];
    }
    //计算两个阈值点的值
    for(int i=0;i<256;i++)
    {
        if(stretch_pl[i]<0.1) //low_value 取接近 10%的总像素的灰度值
        {
            low_value=i;
        }
        if(stretch_pl[i]>0.9) //high_value 取接近 90%的总像素的灰度值
        {
            high_value=i;
            break;
        }
    }
    rate=(float)255/(high_value-low_value+1);
    //进行灰度拉伸
    for(int y=0;y<src->height;y++)
    {
        uchar * ptr_src=(uchar *) (src->imageData+y*src->widthStep);
        uchar * ptr_dst=(uchar *) (dst->imageData+y*dst->widthStep);
        for(int x=0;x<src->width;x++)
        {
            if(ptr_src[x]<low_value)
            {
                ptr_dst[x]=0;
            }
            else if(ptr_src[x]<high_value)
            {
                ptr_dst[x]=(uchar) ((ptr_src[x]-low_value)*rate+0.5);
                if(ptr_dst[x]>255)
                    ptr_dst[x]=high_value;
            }
            else if(ptr_src[x]>=high_value)
            {
                ptr_dst[x]=255;
            }
        }
    }

```

```

    }
}
}

```

## 2. 中值滤波函数

```

void ImageFUNC::Middle_Smooth(IplImage *src, IplImage *dst)
{
    int temp=0;//中间变量
    int flag=0;//循环变量
    int pFilter_Image_Pixel[5]; //窗口像素值
    int mid_pixel_value=0; //中值
    //清空数组并赋初始值为0
    memset(pFilter_Image_Pixel,0,sizeof(pFilter_Image_Pixel));
    //中值滤波
    for(int j=2;j<dst->height-2;j++)
    {
        uchar *ptr_dst=(uchar*)(dst->imageData+j*dst->widthStep);
        for(int i=0;i<dst->width;i++)
        {
            int m=0;
            for(int y=j-2;y<=j+2;y++)
            {
                uchar *ptr_src=(uchar*)(src->imageData+y*src->widthStep);
                for(int x=i;x<=i;x++)
                {
                    pFilter_Image_Pixel[m]=ptr_src[i];
                    m++;
                }
            }
            //把 pFilter_Image_Pixel[m] 中的值按降序排列
            do{
                flag=0;
                for(int m=0;m<4;m++)
                {
                    if(pFilter_Image_Pixel[m]<pFilter_Image_Pixel[m+1])
                    {
                        temp=pFilter_Image_Pixel[m];
                        pFilter_Image_Pixel[m]=pFilter_Image_Pixel[m+1];
                        pFilter_Image_Pixel[m+1]=temp;
                        flag=1;
                    }
                }
            }while(flag==1);
            mid_pixel_value=pFilter_Image_Pixel[2]; //求中值 mid_pixel_value
            ptr_dst[i]=mid_pixel_value; //将中值赋给目标图像的当前点
        }
    }
}

```

### 14.4.2 车牌提取模块

车牌提取采用投影法实现，车牌预处理的系列操作也包含在本模块中。具体代码如下：

```
//系统的变量定义
IplImage *src_image; //原始车辆图像
IplImage *gray_image; //原始灰度图像
IplImage *plate_image; //分割出的车牌图像
IplImage * dst_image[7]; //存放归一化后的字符的图像数组
ImageFUNC m_Func; //图像处理函数对象
myimg m_Cimage; //显示图像对象
ImgThin m_thin; //图像细化对象
//读入原始图像
IplImage *pSrc_Image=NULL;
pSrc_Image=cvCreateImage(cvGetSize(src_image),src_image->depth,
                        src_image->nChannels);
pSrc_Image=cvCloneImage(src_image);

//将图像转化为灰度图像
gray_image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
cvCvtColor(pSrc_Image,gray_image,CV_BGR2GRAY);

//灰度拉伸
//定义拉伸后的图像
IplImage * pStretch_Image=NULL;
pStretch_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
m_Func.Stretch(gray_image,pStretch_Image,0,255);

//图像腐蚀
//定义腐蚀后的图像
IplImage *pErode_Image=NULL;
pErode_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
cvErode(pStretch_Image,pErode_Image,0,1);

//拉伸后的图像减去腐蚀后的图像，得到车牌的轮廓
//定义做差后的图像
IplImage *sub_Image=NULL;
sub_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
cvAbsDiff(pStretch_Image,pErode_Image,sub_Image);

//对图像进行二值化操作
//定义二值化图像
IplImage *pBinary_Image=NULL;
pBinary_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
cvThreshold(sub_Image,pBinary_Image,0,255,CV_THRESH_OTSU);
//对图像进行中值滤波，核采用矩阵(1,1,1,1)T
//定义中值滤波图像
```

## 数字图像处理典型案例详解

```

IplImage *pFilter_Image=NULL;
pFilter_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
//先复制一下要进行中值滤波的图像,因为前两行和最后两行没有进行处理
pFilter_Image=cvCloneImage(pBinary_Image);
m_Func.Middle_Smooth(pBinary_Image,pFilter_Image);

//对二值图像做腐蚀处理,去除噪声
//定义腐蚀后的图像
IplImage* pKernel_Erode_Image=NULL;
pKernel_Erode_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
IplConvKernel * pKernel_Erode=NULL;
pKernel_Erode=cvCreateStructuringElementEx(1,3,0,1,CV_SHAPE_RECT,NULL);
cvErode(pFilter_Image, pKernel_Erode_Image,pKernel_Erode,2);

//对二值图像做膨胀处理,恢复车牌区域
IplConvKernel * pKernel_Dilate=NULL;
pKernel_Dilate=cvCreateStructuringElementEx(1,3,0,1,CV_SHAPE_RECT,NULL);
//定义膨胀后的图像
IplImage* pKernel_Dilate_Image=NULL;
pKernel_Dilate_Image=cvCreateImage(cvGetSize(pSrc_Image),8,1);
cvDilate(pKernel_Erode_Image, pKernel_Dilate_Image,pKernel_Dilate,1);

//对车辆图像做水平投影
//定义像素累加和数组,图像高度不应大于 2048
int level_shadow[2048];
int height=pSrc_Image->height;
int width=pSrc_Image->width;
CvScalar s_shadow;
//清空数组
memset(level_shadow,0,sizeof(level_shadow));

//对图像做水平投影
for(int y=height-1;y>=0;y--)
{
    for(int x=0;x<width;x++)
    {
        s_shadow=cvGet2D(pKernel_Dilate_Image,y,x);
        if(s_shadow.val[0]!=0)
            level_shadow[y]++; //统计每行不为零的像素的个数
    }
}

//对图像水平投影的值做聚类,如果小行数值除以大行数值的商小于 0.6 则认为这两行不是
//一类,将小行数值置为 0
for(int y=height-1;y>=1;y--)
{
    if(level_shadow[y-1]!=0)
    {

```



```

        if((float(level_shadow[y]))/(float(level_shadow[y-1]))<0.6)
            level_shadow[y]=0 ;
    }
}
//统计水平投影中不为零的区间
for(int y=height-1;y>=0;y--)
{
    if(level_shadow[y]!=0)
        level_shadow[y]=level_shadow[y+1]+1;
}
//求出水平投影数组中连续不为零的最大区间, 将此作为车牌的大致高度
int Y_min=0;//车牌高度小坐标
int Y_max=0;//车牌高度大坐标
int M_max_value=0;
M_max_value=level_shadow[0]; //把 level_shadow 的第一个值赋给 M_max_value
for(int y=0;y<height;y++)
{
    if(level_shadow[y]>M_max_value)
    {
        M_max_value=level_shadow[y];
        Y_min=y;
        Y_max=Y_min+M_max_value;
    }
}
if(M_max_value<10)
    AfxMessageBox("提取车牌高度失败!");
//定义 ROI 区域, 切割出车牌的高度
CvRect ROI_Plate_Height;
ROI_Plate_Height.x=0;
ROI_Plate_Height.y=Y_min;
ROI_Plate_Height.width=pSrc_Image->width;
ROI_Plate_Height.height=M_max_value;
cvSetImageROI(pKernel_Dilate_Image,ROI_Plate_Height);

//将此区域复制一份, 便于后续处理
IplImage * pROI_Height_Image=NULL;
pROI_Height_Image=cvCreateImage(cvSize(ROI_Plate_Height.width,ROI_Plate_Height.height),8,1);
cvCopyImage(pKernel_Dilate_Image,pROI_Height_Image);

//对车牌高度区域做闭运算, 得出车牌的矩形区域, 以切割出车牌
//运算核大小采用 (车牌的高度*0.6)
int Copy_M_max_value=M_max_value;
int Close_width=0;
int Close_height=0;

//核大小规定为奇数

```

```

while((Copy_M_max_value%3)!=0)
{
    Copy_M_max_value--;
}
Close_width=int(Copy_M_max_value*0.6);
Close_height=Copy_M_max_value;
IplConvKernel * pKernel_Close=NULL;
pKernel_Close=cvCreateStructuringElementEx(Close_width,Close_height,
        Close_width/2,Close_height/2,CV_SHAPE_RECT,NULL);
cvMorphologyEx(pROI_Height_Image,pROI_Height_Image,NULL,
        pKernel_Close,CV_MOP_CLOSE,1);

//求联通区域的最大宽度,定位车牌的横坐标
int X_min=0;//车牌宽度小坐标
int X_max=0;//车牌宽度大坐标
int M_row_max_value=0;
int count_row[2048];//图像宽度不应大于2048
memset(count_row,0,sizeof(count_row));

//取车牌中间的一条直线进行检测,求此直线上连续不为零的像素的最大宽度,将此作为车牌宽度
int mid_height=M_max_value/2;
uchar*ptr_mid=(uchar*)(pROI_Height_Image->imageData+mid_height
        *pROI_Height_Image->widthStep);
for(int x=width-1;x>=0;x--)
{
    if(ptr_mid[x]!=0)
        count_row[x]=count_row[x+1]+1;
}
//求出 count_row 数组中的最大值
int Max_value_count_row=0;
Max_value_count_row=count_row[0];
for(int x=0;x<width;x++)
{
    if(count_row[x]>Max_value_count_row)
    {
        Max_value_count_row=count_row[x];
        X_min=x;
        X_max=X_min+Max_value_count_row;
    }
}
//车牌的宽度应大于高度的三倍,对切割出的车牌进行验证
if(float(Max_value_count_row)/float(M_max_value)<3||float(Max_value_count_row)/float(M_max_value)>7)
    AfxMessageBox("提取车牌失败!");
//切割出车牌
CvRect ROI_Plate;

```



```

ROI_Plate.x=X_min-3; //对车牌区域做宽度为 3 的放大定位
ROI_Plate.y=Y_min;
ROI_Plate.width=Max_value_count_row+6;
ROI_Plate.height=M_max_value;

//判断车牌定位区域是否合法
if(ROI_Plate.x<0||ROI_Plate.x>width)
    AfxMessageBox("提取车牌失败!");
if(ROI_Plate.y<0|| ROI_Plate.y>height)
    AfxMessageBox("提取车牌失败!");
if((ROI_Plate.x+ROI_Plate.width)>width)
    AfxMessageBox("提取车牌失败!");
if((ROI_Plate.y+ROI_Plate.height)>height)
    AfxMessageBox("提取车牌失败!");
cvSetImageROI(gray_image,ROI_Plate);
plate_image=cvCreateImage(cvSize(ROI_Plate.width,ROI_Plate.height),8,1);
cvCopyImage(gray_image,plate_image);
//释放灰度图像的 ROI 区域
cvResetImageROI(gray_image);

//对车牌图像进行灰度拉伸
m_Func.Stretch(plate_image,plate_image,0,255);
//对车牌图像进行二值化
cvThreshold(plate_image,plate_image,0,255,CV_THRESH_OTSU);
//将车牌图像颜色反转
cvNot(plate_image,plate_image);

```

### 14.4.3 倾斜校正模块

倾斜校正主要是根据车牌字符像素的平均高度来进行的，其代码如下：

```

//图像的倾斜矫正
//定义车牌校正后的图像
IplImage *pRoation=NULL;
pRoation=cvCreateImage(cvGetSize(plate_image),8,1);
CvScalar s,s_new;
double num=0;
double leftaverage=0;
double rightaverage=0;
int iHeight=plate_image->height;
int iWidth=plate_image->width;
double slope=0;
int pix_new;
//计算前半部分斜率
for(int ht=0;ht<iHeight;ht++)
{
    for(int wt=0;wt<iWidth/6;wt++)

```

```

    {
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            num+=iWidth/2-wt;
            leftaverage+=ht*(iWidth/2-wt);
        }
    }
}
leftaverage/=num;
num=0;
//计算后半部分斜率
for(int ht=0;ht<iHeight;ht++)
{
    for(int wt=(iWidth/6)*5;wt<iWidth;wt++)
    {
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            num+=iWidth/2-wt;
            rightaverage+=ht*(iWidth/2-wt);
        }
    }
}
rightaverage/=num;
//求出斜率,并对斜率做适当缩放避免校正过量
slope=(leftaverage-rightaverage)/(iWidth/2)*0.6;
//图像映射
for(int ht=0;ht<iHeight;ht++)
{
    for(int wt=0;wt<iWidth/2;wt++)
    {
        pix_new=int((ht-(wt-iWidth/2)*slope));
        if(pix_new<0||pix_new>=iHeight)
        {
            s.val[0]=255;
            cvSet2D(pRoation,ht,wt,s);
        }
        else
        {
            s=cvGet2D(plate_image,pix_new,wt);
            s_new.val[0]=s.val[0];
            cvSet2D(pRoation,ht,wt,s_new);
        }
    }
}
for(int ht=0;ht<iHeight;ht++)

```

```

{
    for(int wt=iWidth/2;wt<iWidth;wt++)
    {
        pix_new=int((ht-(wt-iWidth/2)*(slope)));
        if(pix_new<0||pix_new>=iHeight)
        {
            s.val[0]=255;
            cvSet2D(pRoation,ht,wt,s);
        }
        else
        {
            s=cvGet2D(plate_image,pix_new,wt);
            s_new.val[0]=s.val[0];
            cvSet2D(pRoation,ht,wt,s);
        }
    }
}
m_Cimage.mSetImg(pRoation);

```

#### 14.4.4 字符分割模块

字符分割模块采用的是逐列扫描图像的方法，根据二值图像的像素值变化来检测字符的左右边界，把字符一个个分割出来。字符分割模块代码如下：

```

//字符分割
//清空用来保存每个字符区域的链表
CRectLink charRect1,charRect2;
charRect1.clear();
charRect2.clear();
int num2=0; //统计第一次切割出的字符的个数
int num_char=0; //统计最终字符的个数
CvScalar s;
int iHeight=plate_image->height;
int iWidth=plate_image->width;
int top; //字符的大致顶部
int bottom; //字符的大致底部

//遇到第一个黑点确定顶部坐标位置
for(int ht=0;ht<iHeight;ht++)
{
    for(int wt=0;wt<iWidth;wt++)
    {
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            //黑点
            top=ht;

```

```

        ht=iHeight;//强制跳出循环
        break;
    }
}
//遇到第一个黑点确定底部坐标位置
for(int ht=iHeight-1;ht>=0;ht--)
{
    for(int wt=0;wt<iWidth;wt++)
    {
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            bottom=ht;//强制跳出循环
            ht=-1;
            break;
        }
    }
}

bool lab=FALSE; //是否进入一个字符分割状态
bool black=FALSE; //扫描时是否发现黑点
CRect rect; // 存放位置信息的结构体
for(int wt=0;wt<iWidth;wt++)
{
    //开始扫描一行
    black=false;
    for(int ht=0;ht<iHeight;ht++)
    {
        // 获得当前像素值
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])//判断是否是黑点
        {
            //如果发现黑点, 设置标志位
            black=true;
            //如果还没有发现黑点, 则进入一个字符分割
            if(lab==false)
            {
                //设置左侧边界
                rect.left=wt;
                //字符分割开始
                lab=true;
            }
            //如果字符分割已经开始了
            else
                //跳出循环
                break;
        }
    }
}

```



```

    }
}
//如果已经扫描到了最右边那列,说明整幅图像扫描完毕
if(wt==(iWidth-1))
    break;
//如果到此时 black 仍为 false,说明扫描了一列都没有发现黑点。表面当前字符分割结束
if((lab==true)&&(black==false))
{
    //将位置信息存入结构体中
    //设置右边界
    rect.right=wt;
    //设置上边界
    rect.top=top;
    //设置下边界
    rect.bottom=bottom;
    //将框外扩一个像素,以免压到字符
    rect.InflateRect(1,1);
    //将这个结构体插入存放位置信息的链表的后面
    charRect1.push_back(rect);
    //设置标志位,开始下一次的字符分割
    lab=false;
    //字符个数加 1
    num2++;
}
//进入下一列的扫描
}

//再将矩形轮廓的 top 和 bottom 精确化
//将链表赋值给链表
charRect2=charRect1;
//将链表的内容清空
charRect2.clear();
//建立一个新的存放位置信息的结构体
CRect rectnew;
//对于链表从头到尾逐个进行扫描
while(!charRect1.empty())
{
    //从链表头上得到一个矩形
    rect=charRect1.front();
    //从链表头上删掉一个
    charRect1.pop_front();
    //判断字符的宽度,以去除中间的小圆点
    if(float((rect.right-rect.left))/float(plate_image->width)>0.1)
    {
        //字符个数加 1
        num_char++;
        //计算更加精确的矩形区域
    }
}

```

```

//获得精确的左边界
rectnew.left=rect.left-1;
//获得精确的右边界
rectnew.right=rect.right-1;
//通过获得的精确左右边界对上下边界重新进行精确定位
//由上至下扫描, 计算上边界
//行
for(int ht=rect.top;ht<rect.bottom;ht++)
{
    //列
    for(int wt=rect.left;wt<rect.right;wt++)
    {
        // 获得当前像素值
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            //设置上边界
            rectnew.top=ht-1;
            //对 ht 进行强制定义以跳出循环
            ht=rect.bottom;
            //跳出循环
            break;
        }
    }
}
//由下至上扫描, 计算下边界
//行
for(int ht=rect.bottom-1;ht>=rect.top;ht--)
{
    //列
    for(int wt=rect.left;wt<rect.right;wt++)
    {
        // 获得当前像素值
        s=cvGet2D(plate_image,ht,wt);
        if(0==s.val[0])
        {
            //设置下边界
            rectnew.bottom=ht+1;
            //对 ht 进行强制定义以跳出循环
            ht=-1;
            //跳出循环
            break;
        }
    }
}
//将得到的新的准确位置信息从后面插入到链表的尾部
charRect2.push_back(rectnew);

```



```

    }
}
//将链表传递给链表
charRect1=charRect2;

//切割图像
CvRect dst_rect[100];
CRect rect2;
for(int i=0;i<num_char;i++)
{
    if(!charRect1.empty())
    {
        rect2=charRect1.front();
        charRect1.pop_front();
        dst_rect[i].x=rect2.left;
        dst_rect[i].y=rect2.top;
        dst_rect[i].width=rect2.right-rect2.left+1;
        dst_rect[i].height=rect2.bottom-rect2.top+1;
    }
}
}

```

#### 14.4.5 字符归一化模块

字符归一化模块的本质是图像大小的缩放，其代码如下：

```

//字符归一化
for(int i=0;i<num_char;i++)
{
    dst_image[i]=cvCreateImage(cvSize(g_width,g_height),8,1);
    cvSetImageROI(plate_image,dst_rect[i]);
    cvNot(plate_image,plate_image);//字符反色处理
    cvResize(plate_image,dst_image[i],CV_INTER_NN);//图像缩放
    m_thin.Thin(dst_image[i],dst_image[i]);//字符细化
    cvResetImageROI(plate_image);
}

```

#### 14.4.6 字符细化模块

细化算法应满足 3 个条件：第一个条件是将条形区域变成一条薄线；第二个条件是薄线应位于原条形区域的中心；第三个条件是薄线应保持原始图像的拓扑特性。图像细化的算法也很多，这里采用的是基于索引表的细化算法。具体代码如下：

```

typedef int MyType;
#define MatType CV_32SC1 //定义数据类型
//*****定义图像细化类*****
class ImgThin
{

```

## 数字图像处理典型案例详解

```
public:
    ImgThin(); //构造函数
    ~ImgThin(); //析构函数
    int Shrink(CvArr* src, CvArr* dst, int iterate = -1); //图像收缩
    IplImage* Shrink(IplImage* src, int iterate = -1);
    int Skel(CvArr* src, CvArr* dst, int iterate = -1); //元素删除
    IplImage* Skel(IplImage* src, int iterate = -1);
    int Thin(CvArr* src, CvArr* dst, int iterate = -1); // 图像细化
    IplImage* Thin(IplImage* src, int iterate = -1);
    int Thresh; //二值化阈值

private:
    void CheckSkel(void);
    void CheckShrink(void);
    void CheckThin(void);
    void CheckTmp(CvMat** tmp, CvSize size); //临时中间变量构造函数
    void Convolution(CvMat* src, CvMat* dst, MyType* kernel); //图像卷积运算
    void CombCopy(CvMat* src, CvMat* dst, CvPoint start); //图像复制
    bool Identical(CvMat* mat1, CvMat* mat2); //判断两个矩阵是否一样
    bool HaveZero(CvMat* src); //检测矩阵中是否有零元素
    void LUT(CvMat* src, CvMat* dst, MyType* lut); //索引表查找
    void Threshold(CvMat* src, CvMat* dst, int Thr, int max); //图像二值化函数
    MyType Kernel[9]; //细化时采用的模板
    CvMat* tmp0; //矩阵定义
    CvMat* tmp1; //矩阵定义
    CvMat* tmp2; //矩阵定义
    CvMat* tmp3; //矩阵定义
    bool use_skel; //定义功能使用标志
    bool use_shrink; //定义功能使用标志
    bool use_thin; //定义功能使用标志
    MyType * skel[8]; //定义功能使用标志
    MyType * shrink;
    MyType * thin[2];
};

//索引表
#define SHRINK_TABLE { \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, \
    1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, \
    1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, \
    1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, \
    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, \
}
```

458

## 数字图像处理典型案例详解

```

static const MyType SKEL4[]={8,26,27,30,31,90,91,94,95};
static const MyType SKEL5[]={4,464,472,496,504};
static const MyType SKEL6[]={8,50,51,54,55,306,307,310,311};
static const MyType SKEL7[]={4,308,310,436,438};
static const MyType SKEL8[]={8,176,180,240,244,432,436,496,500};

#define KERNEL_TABLE {1, 8, 64,\
                      2, 16,128,\
                      4, 32, 256} //细化模板表
//*****各成员函数实现*****
ImgThin::ImgThin() //构造函数, 初始化类成员变量
{
    use_shrink=false;
    use_skel=false;
    use_thin=false;
    MyType kernel[]=KERNEL_TABLE;
    memcpy(Kernel,kernel,sizeof(kernel));
    tmp0=tmp1=tmp2=tmp3=NULL;
    Thresh=128;
}
//析构函数, 释放内存
ImgThin::~ImgThin()
{
    if(use_shrink)
        delete [] shrink;
    if(use_skel)
    {
        for(int i=0; i<8; i++)
            delete [] skel[i];
    }
    if(use_thin)
    {
        delete [] thin[0];
        delete [] thin[1];
    }
    if(tmp0)
        cvReleaseMat(&tmp1);
    if(tmp1)
        cvReleaseMat(&tmp1);
    if(tmp2)
        cvReleaseMat(&tmp2);
    if(tmp3)
        cvReleaseMat(&tmp3);
}
//判断两个矩阵是否相同
bool ImgThin::Identical(CvMat* mat1, CvMat* mat2)

```

```

{
    int i,j;
    int rows=mat1->rows;
    int cols=mat1->cols;
    for(i=0; i<rows; i++){
        MyType* ptr1=(MyType*)(mat1->data.ptr+mat1->step*i);
        MyType* ptr2=(MyType*)(mat2->data.ptr+mat2->step*i);
        for(j=0; j<cols; j++)
        {
            if(*ptr1++ != *ptr2++)
                return false;
        }
    }
    return true;
}
//检测矩阵中是否有零元素,并返回结果
bool ImgThin::HaveZero(CvMat* src)
{
    int i,j;
    int rows=src->rows;
    int cols=src->cols;
    for(i=0; i<rows; i++){
        MyType* ptr1 = (MyType*)(src->data.ptr+src->step*i);
        for(j=0; j< cols; j++)
        {
            if(*ptr1++ == 0)
                return true;
        }
    }
    return false;
}
//检测是否调用过 Tmp 函数,若没有则对变量进行赋值
void ImgThin::CheckTmp(CvMat** tmp, CvSize size)
{
    CvMat* Tmp=*tmp;
    if(Tmp==NULL)
    {
        *tmp=cvCreateMat(size.height,size.width,MatType);
        return;
    }
    CvSize size_tmp=cvGetSize(Tmp);
    if((size_tmp.height != size.height)|| (size_tmp.width != size.width))
    {
        cvReleaseMat(tmp);
        *tmp=cvCreateMat(size.height,size.width,MatType);
    }
}

```

## 数字图像处理典型案例详解

```

//检测是否调用过 Shrink 函数, 若没有则对变量进行赋值
void ImgThin::CheckShrink(void)
{
    if(!use_shrink)
    {
        MyType table[]=SHRINK_TABLE;
        use_shrink=true;
        shrink=new MyType[512];
        memcpy(shrink,&table,512*sizeof(MyType));
        for(int i=0;i<512;i++)
            shrink[i]=!shrink[i];
    }
}

//图像卷积运算函数
void ImgThin::Convolution(CvMat* src, CvMat* dst, MyType* kernel)
{
    int i,j,m,n;
    int rows=src->rows;
    int cols=src->cols;
    for(i=1;i<rows-1;i++)
    {
        MyType* ptr0=(MyType*)src->data.ptr+cols*i+1;
        MyType* ptr1=(MyType*)dst->data.ptr+cols*i+1;
        for(j=1;j<cols-1;j++)
        {
            *ptr1=0;
            for(m=-1;m<=1;m++)
            {
                for(n=-1;n<=1;n++)
                {
                    MyType* ptr2=ptr0+cols*m+n;
                    *ptr1+=(*ptr2)*kernel[3*(m+1)+(n+1)];
                    if(*ptr1<0)
                        int a=1;
                }
            }
            ptr0++;
            ptr1++;
        }
    }
}

//索引表查找函数
void ImgThin::LUT(CvMat* src, CvMat* dst, MyType* lut)
{
    int i,j;
    int rows=src->height;
    int cols=src->width;

```



```

    for(i=0;i<rows;i++)
    {
        MyType* ptr0=(MyType*)(src->data.ptr+src->step*i);
        MyType* ptr1=(MyType*)(dst->data.ptr+dst->step*i);
        for(j=0;j<cols;j++)
        {
            *ptr1=lut[*ptr0];
            ptr1++;
            ptr0++;
        }
    }
}
//矩阵复制函数
void ImgThin::CombCopy(CvMat* src, CvMat* dst, CvPoint start)
{
    int i,j;
    int rows=src->rows;
    int cols=src->cols;
    for(i=start.y;i<rows;i+=2)
    {
        MyType* ptr0=(MyType*)(src->data.ptr+src->step*i)+start.x;
        MyType* ptr1=(MyType*)(dst->data.ptr+dst->step*i)+start.x;
        for(j=start.x;j<cols;j+=2)
        {
            *ptr1=*ptr0;
            ptr0+=2;
            ptr1+=2;
        }
    }
}
//图像收缩函数
int ImgThin::Shrink(CvArr* src,CvArr* dst, int iterate)
{
    CheckShrink();
    CvSize size = cvGetSize(src);
    CheckTmp(&tmp0,size);
    cvConvert(src,tmp0);
    size.height = size.height + 2;
    size.width = size.width + 2;
    CheckTmp(&tmp1,size);
    CheckTmp(&tmp2,size);
    cvRectangle(tmp2,cvPoint(0,0),cvPoint(size.width-1,size.height-1),cvScalar(0));
    CheckTmp(&tmp3,size);
    cvCopyMakeBorder(tmp0,tmp1,cvPoint(1,1),IPL_BORDER_REPLICATE);
    Threshold(tmp1,tmp1,Thresh,1);
    while((iterate> 0)|| (iterate<0))
    {

```

```

        if(iterate >0)
            iterate--;
        cvCopy(tmp1,tmp3);
        //First subiteration
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,shrink);
        cvAnd(tmp1,tmp2,tmp2);
        CombCopy(tmp2,tmp1,cvPoint(1,1));
        //Second subiteration
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,shrink);
        cvAnd(tmp1,tmp2,tmp2);
        CombCopy(tmp2,tmp1,cvPoint(2,2));
        //Third subiteration
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,shrink);
        cvAnd(tmp1,tmp2,tmp2);
        CombCopy(tmp2,tmp1,cvPoint(2,1));
        //Fourth subiteration
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,shrink);
        cvAnd(tmp1,tmp2,tmp2);
        CombCopy(tmp2,tmp1,cvPoint(1,2));
        if( Identical(tmp3,tmp1))
            break;
    }
    CvMat mask;
    cvGetSubRect(tmp1,&mask,cvRect(1,1,size.width-2,size.height-2));
    Threshold(tmp1,tmp1,0,255);
    cvAnd(tmp0,&mask,tmp0);
    cvConvert(tmp0,dst);
    return 0;
}

IplImage* ImgThin::Shrink(IplImage* src, int iterate)
{
    IplImage* dst=cvCloneImage(src);
    Shrink(src,dst,iterate);
    return dst;
}

//根据模板对图像元素进行删除
void ImgThin::CheckSkel(void)
{
    int i,j;
    if(!use_skel)
    {
        use_skel = true;
        for(i=0; i<8; i++)

```

```

    {
        skel[i] = new MyType[512];
        for(j=0; j<512; j++)
            skel[i][j] = 1;
    }
    for(i=1; i<=SKEL1[0]; i++)
        skel[0][SKEL1[i]]=0;
    for(i=1; i<=SKEL2[0]; i++)
        skel[1][SKEL2[i]]=0;
    for(i=1; i<=SKEL3[0]; i++)
        skel[2][SKEL3[i]]=0;
    for(i=1; i<=SKEL4[0]; i++)
        skel[3][SKEL4[i]]=0;
    for(i=1; i<=SKEL5[0]; i++)
        skel[4][SKEL5[i]]=0;
    for(i=1; i<=SKEL6[0]; i++)
        skel[5][SKEL6[i]]=0;
    for(i=1; i<=SKEL7[0]; i++)
        skel[6][SKEL7[i]]=0;
    for(i=1; i<=SKEL8[0]; i++)
        skel[7][SKEL8[i]]=0;
    }
}
//图像二值化函数
void ImgThin::Threshold(CvMat* src, CvMat* dst, int Thr, int max)
{
    int i,j;
    int rows = src->rows;
    int cols = src->cols;
    for(i=0; i<rows; i++) //对图像所有元素进行逐个判断
    {
        MyType * ptr0=(MyType *) (src->data.ptr) + cols* i;
        MyType* ptr1=(MyType *) (dst->data.ptr) + cols*i;
        for(j=0; j<cols; j++)
        {
            if(*ptr0>Thr)
                *ptr1=max;
            else
                *ptr1=0;
            ptr0++;
            ptr1++;
        }
    }
}
//元素删除算法函数
int ImgThin::Skel(CvArr* src, CvArr* dst, int iterate)
{

```

```

    CheckSkel();
    CvSize size = cvGetSize(src);
    CheckTmp(&tmp0, size);
    cvConvert(src, tmp0);
    size.height = size.height + 2;
    size.width = size.width + 2;
    CheckTmp(&tmp1, size);
    CheckTmp(&tmp2, size);
    cvRectangle(tmp2, cvPoint(0, 0), cvPoint(size.width-1, size.height-1), cvScalar(0));
    CheckTmp(&tmp3, size);
    cvCopyMakeBorder(tmp0, tmp1, cvPoint(1, 1), IPL_BORDER_REPLICATE);
    Threshold(tmp1, tmp1, Thresh, 1);
    while( (iterate > 0) || (iterate < 0) )
    {
        if(iterate > 0)
            iterate--;
        cvCopy(tmp1, tmp3);
        for(int i=0; i<8; i++)
        {
            Convolution(tmp1, tmp2, Kernel);
            LUT(tmp2, tmp2, skel[i]);
            cvAnd(tmp1, tmp2, tmp1);
        }
        if(Identical(tmp3, tmp1))
            break;
    }
    CvMat mask;
    cvGetSubRect(tmp1, &mask, cvRect(1, 1, size.width-2, size.height-2));
    Threshold(tmp1, tmp1, 0, 255);
    cvAnd(tmp0, &mask, tmp0);
    cvConvert(tmp0, dst);
    return 0;
}

IplImage* ImgThin::Skel(IplImage* src, int iterate)
{
    IplImage* dst= cvCloneImage(src);
    Skel(src, dst, iterate);
    return dst;
}
//检测是否调用过 Thin 函数, 若没有则对变量进行赋值
void ImgThin::CheckThin(void)
{
    if(!use_thin){
        MyType table1[512]=THIN_TABLE1;
        MyType table2[512]=THIN_TABLE2;
        use_thin=true;
    }
}

```



```

        thin[0]=new MyType[512];
        thin[1]=new MyType[512];
        memcpy(thin[0],&table1,512*sizeof(MyType));
        memcpy(thin[1],&table2,512*sizeof(MyType));
    }
}
//图像细化过程函数
int ImgThin::Thin(CvArr* src,CvArr* dst, int iterate)
{
    CheckThin();//判断是否细化过
    CvSize size = cvGetSize(src);//获得图像尺寸
    CheckTmp(&tmp0,size);
    cvConvert(src,tmp0);
    size.height = size.height + 2; //由于模板是 3×3 的, 故对图像高度做加 2 处理
    size.width = size.width + 2; //由于模板是 3×3 的, 故对图像宽度做加 2 处理
    CheckTmp(&tmp1,size);
    CheckTmp(&tmp2,size);
    cvRectangle(tmp2,cvPoint(0,0),cvPoint(size.width-1,size.height-1),cvScalar(0));
    CheckTmp(&tmp3,size);
    cvCopyMakeBorder(tmp0,tmp1,cvPoint(1,1),IPL_BORDER_REPLICATE);
    Threshold(tmp1,tmp1,Thresh,1); //二值化图像
    while((iterate>0)|| (iterate<0)) //进行反复迭代运算
    {
        if(iterate>0)
            iterate--;
        cvCopy(tmp1,tmp3);
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,thin[0]);
        cvAnd(tmp1,tmp2,tmp1);
        Convolution(tmp1,tmp2,Kernel);
        LUT(tmp2,tmp2,thin[1]);
        cvAnd(tmp1,tmp2,tmp1);
        if(Identical(tmp3,tmp1))
            break;
    }

    CvMat mask;
    cvGetSubRect(tmp1,&mask,cvRect(1,1,size.width-2,size.height-2)); //获得子区域
    Threshold(tmp1,tmp1,0,255); //二值化
    cvAnd(tmp0,&mask,tmp0); //和掩码进行与运算
    cvConvert(tmp0,dst);
    return 0;
}

//图像细化函数
IplImage* ImgThin::Thin(IplImage* src, int iterate)
{

```

```

    IplImage* dst= cvCloneImage(src); //复制原始图像
    Thin(src,dst,iterate); //图像细化
    return dst; //返回目标图像
}

```

#### 14.4.7 字符特征提取模块

字符特征提取模块主要是提取归一化和细化后字符的特征，将这些特征和模板中字符的特征进行匹配，选择最相近的匹配作为分类结果。其代码如下：

```

//定义结构体
struct pattern
{
    double feature[33]; //样本的特征向量
    int number;          //待识别字符在样本库中的序列号
};

//特征提取函数
void GetFeature(IplImage* src,pattern &pat)
{
    CvScalar s;
    int i,j;
    for(i=0;i<33;i++)
        pat.feature[i]=0.0;
    //图像大小是 20×40，分成为 25 块 (5×5)
    //第 1 行，第 1 块
    for(j=0;j<8;j++)
    {
        for(i=0;i<4;i++)
        {
            s=cvGet2D(src,j,i);
            if(s.val[0]==255)
                pat.feature[0]+=1.0;
        }
    }

    //其他 25 块的处理代码类似，在此略去。

    //统计方向交点特征
    for(i=0;i<20;i++)
    {
        s=cvGet2D(src,8,i);
        if(s.val[0]==255)
            pat.feature[25]+=1.0;
    }
    for(i=0;i<20;i++)
    {

```



```

        s=cvGet2D(src,16,i);
        if(s.val[0]==255)
            pat.feature[26]+=1.0;
    }
    for(i=0;i<20;i++)
    {
        s=cvGet2D(src,24,i);
        if(s.val[0]==255)
            pat.feature[27]+=1.0;
    }
    for(i=0;i<20;i++)
    {
        s=cvGet2D(src,32,i);
        if(s.val[0]==255)
            pat.feature[28]+=1.0;
    }
    for(j=0;j<40;j++)
    {
        s=cvGet2D(src,j,4);
        if(s.val[0]==255)
            pat.feature[29]+=1.0;
    }
    for(j=0;j<40;j++)
    {
        s=cvGet2D(src,j,8);
        if(s.val[0]==255)
            pat.feature[30]+=1.0;
    }
    for(j=0;j<40;j++)
    {
        s=cvGet2D(src,j,12);
        if(s.val[0]==255)
            pat.feature[31]+=1.0;
    }
    for(j=0;j<40;j++)
    {
        s=cvGet2D(src,j,16);
        if(s.val[0]==255)
            pat.feature[32]+=1.0;
    }
}

```

#### 14.4.8 车牌字符识别模块

车牌识别系统采用的是模板匹配的方法来识别车牌字符，作为示例，只提取了部分的省份简称汉字模板，包括：川、鄂、黑、京、辽、琼、湘、粤、浙。车牌字符的后六个字符是 0~9 和 A~Z（除去 I 和 O）中字符的组合。系统在进行车牌字符识别时，先载入事先提取的车牌字符图

像模板, 然后提取字符特征, 最后进行识别。其代码如下:

```

IplImage * char_sample[34]; // 字符样本图像数组
IplImage * hanzi_sample[9]; // 汉字样本图像数组
pattern char_pattern[34]; // 定义字符样品库结构数组
pattern hanzi_pattern[9]; // 定义汉字样品库结构数组
pattern TestSample[7]; // 定义待识别字符结构数组
// 载入字符模板
char_sample[0]=cvLoadImage("字符模板\\0.bmp", 0);
char_sample[1]=cvLoadImage("字符模板\\1.bmp", 0);
char_sample[2]=cvLoadImage("字符模板\\2.bmp", 0);
char_sample[3]=cvLoadImage("字符模板\\3.bmp", 0);
char_sample[4]=cvLoadImage("字符模板\\4.bmp", 0);
char_sample[5]=cvLoadImage("字符模板\\5.bmp", 0);
char_sample[6]=cvLoadImage("字符模板\\6.bmp", 0);
char_sample[7]=cvLoadImage("字符模板\\7.bmp", 0);
char_sample[8]=cvLoadImage("字符模板\\8.bmp", 0);
char_sample[9]=cvLoadImage("字符模板\\9.bmp", 0);
char_sample[10]=cvLoadImage("字符模板\\A.bmp", 0);
char_sample[11]=cvLoadImage("字符模板\\B.bmp", 0);
char_sample[12]=cvLoadImage("字符模板\\C.bmp", 0);
char_sample[13]=cvLoadImage("字符模板\\D.bmp", 0);
char_sample[14]=cvLoadImage("字符模板\\E.bmp", 0);
char_sample[15]=cvLoadImage("字符模板\\F.bmp", 0);
char_sample[16]=cvLoadImage("字符模板\\G.bmp", 0);
char_sample[17]=cvLoadImage("字符模板\\H.bmp", 0);
char_sample[18]=cvLoadImage("字符模板\\J.bmp", 0);
char_sample[19]=cvLoadImage("字符模板\\K.bmp", 0);
char_sample[20]=cvLoadImage("字符模板\\L.bmp", 0);
char_sample[21]=cvLoadImage("字符模板\\M.bmp", 0);
char_sample[22]=cvLoadImage("字符模板\\N.bmp", 0);
char_sample[23]=cvLoadImage("字符模板\\P.bmp", 0);
char_sample[24]=cvLoadImage("字符模板\\Q.bmp", 0);
char_sample[25]=cvLoadImage("字符模板\\R.bmp", 0);
char_sample[26]=cvLoadImage("字符模板\\S.bmp", 0);
char_sample[27]=cvLoadImage("字符模板\\T.bmp", 0);
char_sample[28]=cvLoadImage("字符模板\\U.bmp", 0);
char_sample[29]=cvLoadImage("字符模板\\V.bmp", 0);
char_sample[30]=cvLoadImage("字符模板\\W.bmp", 0);
char_sample[31]=cvLoadImage("字符模板\\X.bmp", 0);
char_sample[32]=cvLoadImage("字符模板\\Y.bmp", 0);
char_sample[33]=cvLoadImage("字符模板\\Z.bmp", 0);
// 载入汉字模板
hanzi_sample[0]=cvLoadImage("字符模板\\川.bmp", 0);
hanzi_sample[1]=cvLoadImage("字符模板\\鄂.bmp", 0);
hanzi_sample[2]=cvLoadImage("字符模板\\黑.bmp", 0);
hanzi_sample[3]=cvLoadImage("字符模板\\京.bmp", 0);

```

```

hanzi_sample[4]=cvLoadImage("字符模板\\辽.bmp",0);
hanzi_sample[5]=cvLoadImage("字符模板\\琼.bmp",0);
hanzi_sample[6]=cvLoadImage("字符模板\\湘.bmp",0);
hanzi_sample[7]=cvLoadImage("字符模板\\粤.bmp",0);
hanzi_sample[8]=cvLoadImage("字符模板\\浙.bmp",0);
//提取字符样本特征
for(int i=0;i<34;i++)
{
    GetFeature(char_sample[i],char_pattern[i]);
}
//提取汉字字符特征
for(int i=0;i<9;i++)
{
    GetFeature(hanzi_sample[i],hanzi_pattern[i]);
}
//提取待识别字符特征
for(int i=0;i<7;i++)
{
    GetFeature(dst_image[i],TestSample[i]);
}
//进行模板匹配
double min=100000.0;
for(int num=0;num<1;num++)
{
    for(int i=0;i<9;i++)
    {
        double diff=0.0;
        for(int j=0;j<25;j++)
        {
            diff+=fabs(TestSample[num].feature[j]-hanzi_pattern[i].feature[j]);
        }
        for(int j=25;j<33;j++)
        {
            diff+=fabs(TestSample[num].feature[j]-hanzi_pattern[i].feature[j])*9;
        }
        if(diff<min)
        {
            min=diff;
            TestSample[num].number=i;
        }
    }
}
for(int num=1;num<7;num++)
{
    double min_min=1000000.0;
    for(int i=0;i<34;i++)
    {

```

```
double diff_diff=0.0;
for(int j=0;j<25;j++)
{
    diff_diff+=fabs(TestSample[num].feature[j]-char_pattern[i].
                    feature[j]);
}
for(int j=25;j<33;j++)
{
    diff_diff+=fabs(TestSample[num].feature[j]-char_pattern[i].
                    feature[j]);
}
if(diff_diff<min_min)
{
    min_min=diff_diff;
    TestSample[num].number=i;
}
}
}
CString result=""; //存放识别出的字符
for(int i=0;i<1;i++)
{
    switch (TestSample[i].number)
    {
    case 0:
        result+="川";
        break;
    case 1:
        result+="鄂";
        break;
    case 2:
        result+="黑";
        break;
    case 3:
        result+="京";
        break;
    case 4:
        result+="辽";
        break;
    case 5:
        result+="琼";
        break;
    case 6:
        result+="湘";
        break;
    case 7:
        result+="粤";
        break;
    }
```

```
        case 8:
            result+="浙";
            break;
        default:
            AfxMessageBox("识别失败");
            break;
    }
}
for(int i=1;i<7;i++)
{
    switch (TestSample[i].number)
    {
        case 0:
            result+="0";
            break;
        case 1:
            result+="1";
            break;
        case 2:
            result+="2";
            break;
        case 3:
            result+="3";
            break;
        case 4:
            result+="4";
            break;
        case 5:
            result+="5";
            break;
        case 6:
            result+="6";
            break;
        case 7:
            result+="7";
            break;
        case 8:
            result+="8";
            break;
        case 9:
            result+="9";
            break;
        case 10:
            result+="A";
            break;
        case 11:
            result+="B";
            break;
    }
```



```
case 12:
    result+="C";
    break;
case 13:
    result+="D";
    break;
case 14:
    result+="E";
    break;
case 15:
    result+="F";
case 16:
    result+="G";
    break;
case 17:
    result+="H";
    break;
case 18:
    result+="J";
    break;
case 19:
    result+="K";
    break;
case 20:
    result+="L";
    break;
case 21:
    result+="M";
    break;
case 22:
    result+="N";
    break;
case 23:
    result+="P";
    break;
case 24:
    result+="Q";
    break;
case 25:
    result+="R";
    break;
case 26:
    result+="S";
    break;
case 27:
    result+="T";
    break;
case 28:
```



```
        result+="U";
        break;
    case 29:
        result+="U";
        break;
    case 30:
        result+="W";
        break;
    case 31:
        result+="X";
        break;
    case 32:
        result+="Y";
        break;
    case 33:
        result+="Z";
        break;
    default:
        AfxMessageBox("识别失败");
        break;
    }
}
AfxMessageBox(result);//显示结果
}
```

## 14.5 经验分享

在实际应用开发中，需要注意以下问题：

1) 关于车牌的定位。车牌定位在整个系统中的地位非常重要，如果定位不准确，后面的识别过程就无从谈起。为了能够更精确地定位车牌，可以根据车牌自身的某些特征，加强图像的预处理，对图像多次进行平滑、去噪及二次定位等操作。

2) 关于车牌的规格及车牌字符模板。中华人民共和国公共安全行业标准《中华人民共和国机动车号牌》(GA36—2007) 对车牌的规格及车牌字符做了明确的规定。在实际项目开发过程中，要充分利用该标准文件中的规格规定作为车牌检测定位的先验信息。同时，建立的车牌字符模板库，要涵盖标准文件中规定的全部字样。

## 参 考 文 献

- [1] 刘海波, 沈晶, 郭耸, 等. Visual C++数字图像处理技术详解[M]. 北京: 机械工业出版社, 2010.
- [2] 李昊, 傅曦. 精通 Visual C++指纹模式识别系统算法及实现[M]. 北京: 人民邮电出版社, 2008.
- [3] 求是科技, 张宏林. 精通 Visual C++数字图像处理模式识别技术及工程实践[M]. 2 版. 北京: 人民邮电出版社, 2008.
- [4] 张宏林. 精通 Visual C++数字图像处理典型算法及实现[M]. 北京: 人民邮电出版社, 2008.
- [5] 张宏林. Visual C++数字图像模式识别技术及工程实践[M]. 北京: 人民邮电出版社, 2003.
- [6] 王占全, 徐慧. 精通 Visual C++数字图像处理技术与工程案例[M]. 北京: 人民邮电出版社, 2009.
- [7] 钟志光, 卢君, 刘伟荣. Visual C++ . NET 数字图像处理实例与解析[M]. 北京: 清华大学出版社, 2003.
- [8] 苏彦华, 等. Visual C++数字图像识别技术典型案例[M]. 北京: 人民邮电出版社, 2004.
- [9] 左飞, 万晋森, 刘航. Visual C++数字图像处理开发入门与编程实践[M]. 北京: 电子工业出版社, 2008.
- [10] 求是科技. Visual C++数字图像处理典型算法及实现[M]. 北京: 人民邮电出版社, 2006.
- [11] 曲扬. 精通 Visual C++实效编程 280 例[M]. 北京: 人民邮电出版社, 2009.
- [12] 谢凤英, 赵丹培, 等. Visual C++数字图像处理[M]. 北京: 电子工业出版社, 2008.
- [13] 王颜容. 指纹图像分割与匹配算法研究[D]. 济南: 山东大学硕士学位论文, 2005.
- [14] 师忠超. 指纹图像分析及特征提取[D]. 北京: 中国科学院博士学位论文, 2005.
- [15] 王玮. 自动指纹识别系统关键技术研究[D]. 重庆大学博士学位论文, 2007.
- [16] 陈锻生, 刘政凯. 肤色检测技术综述[J]. 计算机学报, 2006, 29(2): 194-207.
- [17] 郑庆. 基于肤色的人脸检测与人眼定位[D]. 成都: 电子科技大学硕士学位论文, 2007.
- [18] 袁金国. 遥感图像数字处理[M]. 北京: 中国环境科学出版社, 2006.
- [19] 汤国安. 遥感数字图像处理[M]. 北京: 科学出版社, 2004.
- [20] 汤竞煌, 聂智龙. 遥感图像的几何校正[J]. 测绘与空间地理信息, 2007:30(2):100-102, 106.

- [21] 刘瑞祯, 谭铁牛. 数字图像水印研究综述[J]. 电子学报, 2000(8):39-48.
- [22] 张春田, 苏育挺, 张静. 数字图像压缩编码[M]. 北京: 清华大学出版社, 2006.
- [23] 范慧赞. CT 图像滤波反投影重建算法的研究[D]. 西安: 西北工业大学硕士学位论文, 2007.
- [24] 李志鹏, 丛鹏, 邬海峰. 代数迭代算法进行 CT 图像重建的研究[C]. 核电子学与探测技术, 2005(2):184-186.
- [25] 陈胜勇, 刘盛. 基于 OpenCV 的计算机视觉技术实现[M]. 北京: 科学出版社, 2008.
- [26] 于仕琪. OpenCV 教程: 基础篇[M]. 北京: 北京航空航天大学出版社, 2007.



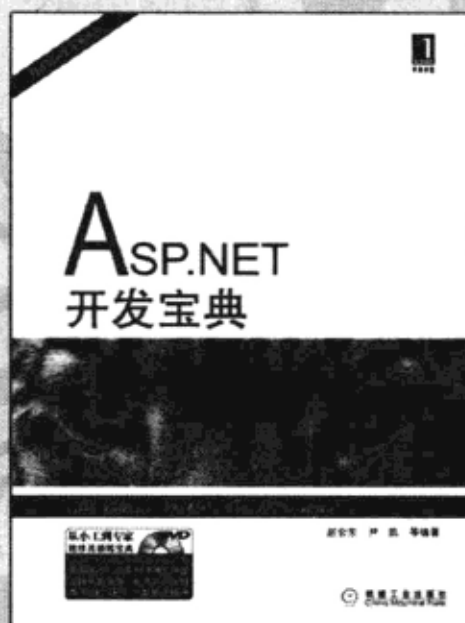
C语言开发宝典  
书号：978-7-111-37786-3  
定价：59.00元



Java开发宝典  
书号：978-7-111-37847-1  
定价：99.00元



C#开发宝典  
书号：978-7-111-37905-8  
定价：89.00元



ASP.NET开发宝典  
书号：978-7-111-37923-2  
定价：89.00元



PHP开发宝典  
书号：978-7-111-38093-1  
定价：89.00元



Visual C++开发宝典  
书号：978-7-111-38001-6  
定价：99.00元



Android开发宝典  
书号：978-7-111-37933-1  
定价：79.00元



专业成就人生  
立体服务大众

www.hzbook.com

## 填写读者调查表 加入华章书友会 获赠精彩技术书 参与活动和抽奖

尊敬的读者：

感谢您选择华章图书。为了聆听您的意见，以便我们能够为您提供更优秀的图书产品，敬请您抽出宝贵的时间填写本表，并按底部的地址邮寄给我们（您也可通过www.hzbook.com填写本表）。您将加入我们的“华章书友会”，及时获得新书资讯，免费参加书友会活动。我们将定期选出若干名热心读者，免费赠送我们出版的图书。请一定填写书名书号并留全您的联系信息，以便我们联络您，谢谢！

书名： 书号： 7-111-( )

姓名：	性别： <input type="checkbox"/> 男 <input type="checkbox"/> 女	年龄：	职业：
通信地址：		E-mail：	
电话：	手机：	邮编：	

1. 您是如何获知本书的：

☐ 朋友推荐 ☐ 书店 ☐ 图书目录 ☐ 杂志、报纸、网络等 ☐ 其他

2. 您从哪里购买本书：

☐ 新华书店 ☐ 计算机专业书店 ☐ 网上书店 ☐ 其他

3. 您对本书的评价是：

技术内容	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
文字质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
版式封面	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
印装质量	<input type="checkbox"/> 很好	<input type="checkbox"/> 一般	<input type="checkbox"/> 较差	<input type="checkbox"/> 理由_____
图书定价	<input type="checkbox"/> 太高	<input type="checkbox"/> 合适	<input type="checkbox"/> 较低	<input type="checkbox"/> 理由_____

4. 您希望我们的图书在哪些方面进行改进？

5. 您最希望我们出版哪方面的图书？如果有英文版请写出书名。

6. 您有没有写作或翻译技术图书的想法？

☐ 是，我的计划是\_\_\_\_\_ ☐ 否

7. 您希望获取图书信息的形式：

☐ 邮件 ☐ 信函 ☐ 短信 ☐ 其他\_\_\_\_\_

请寄：北京市西城区百万庄南街1号 机械工业出版社 华章公司 计算机图书策划部收

邮编：100037 电话：(010) 88379512 传真：(010) 68311602 E-mail: hzjsj@hzbook.com



# Visual C++

## 数字图像处理典型案例详解

Detailed Explanation of Visual C++ Digital Image Processing with Typical Cases

本书以Visual C++数字图像处理技术为主线，结合典型的图像系统开发案例，按照从理论、设计到实现的过程详细进行剖析讲解。案例从应用角度涉及娱乐、文化、医疗、交通、遥感、安防、司法等多个典型应用领域，从技术角度涉及数字图像的文件读写、显示、编辑、滤镜增效、压缩编解码、几何变换、灰度变换、色彩空间变换、特征变换、增强、分割、复原、配准、检索、重建、形态学处理、运动目标检测、跟踪、识别等，几乎涵盖了数字图像处理的整个技术领域及部分模式识别内容，同时还介绍了OpenCV和VTK等开发环境及其与Visual C++联合开发的实用技术。在每个案例的最后，还与读者分享了开发经验。本书配有全部案例的完整源程序，便于读者学习和在实际开发中使用。

本书案例丰富，工程性强；解读深入，技术性强；代码完整，实用性强；抛砖引玉，启发性强。

本书适合从事计算机视觉、数字图像处理、模式识别相关工作的研究人员、工程技术人员，以及相关专业的教师和学生阅读参考。



CD-ROM

客服热线:(010) 88378991, 88361066  
 购书热线:(010) 68326294, 88379649, 68995259  
 投稿热线:(010) 88379604  
 读者信箱:hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书 [www.china-pub.com](http://www.china-pub.com)



上架指导：计算机/程序设计/Visual C++

ISBN 978-7-111-38871-5



9 787111 388715

定价：69.00元（附光盘）